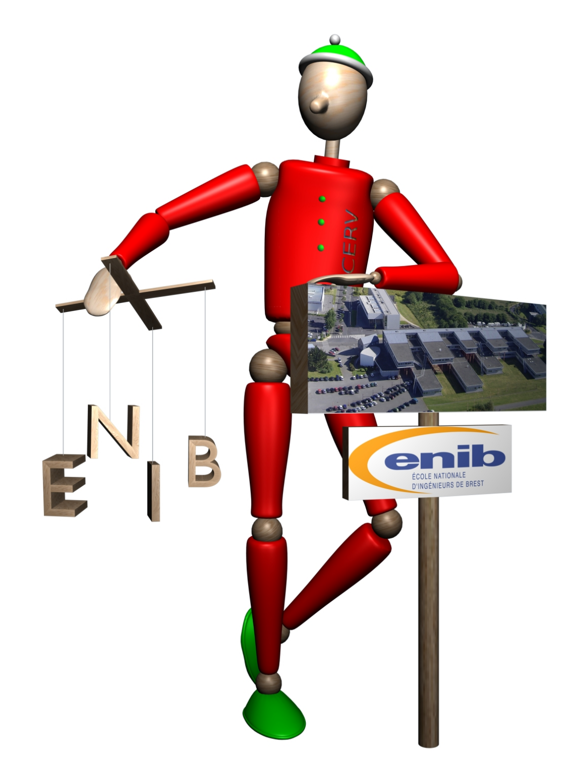


— Cours d'Informatique S1 —

Initiation à l'algorithmique

Jacques TISSEAU

LISYC EA 3883 UBO-ENIB-ENSIETA
Centre Européen de Réalité Virtuelle
tisseau@enib.fr



Ces notes de cours accompagnent les enseignements d'informatique du 1er semestre (S1) de l'Ecole Nationale d'Ingénieurs de Brest (ENIB : www.enib.fr). Leur lecture ne dispense en aucun cas d'une présence attentive aux cours ni d'une participation active aux travaux dirigés.

Avec la participation de Romain Bénard, Stéphane Bonneaud, Cédric Buche, Gireg Desmeulles, Eric Maisel, Aléxis Nédélec, Marc Parenthoën et Cyril Septseault.

version du 01 septembre 2009

Sommaire

« Chaque programme d'ordinateur est un modèle, forgé par l'esprit, d'un processus réel ou imaginaire. Ces processus, qui naissent de l'expérience et de la pensée de l'homme, sont innombrables et complexes dans leurs détails. A tout moment, ils ne peuvent être compris que partiellement. Ils ne sont que rarement modélisés d'une façon satisfaisante dans nos programmes informatiques. Bien que nos programmes soient des ensembles de symboles ciselés avec soin, des mosaïques de fonctions entrecroisées, ils ne cessent d'évoluer. Nous les modifions au fur et à mesure que notre perception du modèle s'approfondit, s'étend et se généralise, jusqu'à atteindre un équilibre métastable aux frontières d'une autre modélisation possible du problème. L'ivresse joyeuse qui accompagne la programmation des ordinateurs provient des allers-retours continuels, entre l'esprit humain et l'ordinateur, des mécanismes exprimés par des programmes et de l'explosion de visions nouvelles qu'ils apportent. Si l'art traduit nos rêves, l'ordinateur les réalise sous la forme de programmes ! »

Abelson H., Sussman G.J. et Sussman J., [1]

Chapitre 1

Introduction générale



Sommaire

1.1 Contexte scientifique

1.1.1 Informatique

Le terme INFORMATIQUE est un néologisme proposé en 1962 par Philippe Dreyfus pour caractériser le traitement automatique de l'information : il est construit sur la contraction de l'expression « information automatique ». Ce terme a été accepté par l'Académie française en avril 1966, et l'informatique devint alors officiellement la science du traitement automatique de l'information, où l'information est considérée comme le support des connaissances humaines et des communications dans les domaines techniques, économiques et sociaux (figure 1.1). Le mot INFORMATIQUE n'a pas vraiment d'équivalent aux Etats-Unis où l'on parle de *Computing Science* (science du calcul) alors que *Informatics* est admis par les Britanniques.

Définition 1.1 : INFORMATIQUE
L'informatique est la science du traitement automatique de l'information.

L'informatique traite de deux aspects complémentaires : les programmes immatériels (logiciel, *software*) qui décrivent un traitement à réaliser et les machines (matériel, *hardware*) qui exécutent ce traitement. Le matériel est donc l'ensemble des éléments physiques (microprocesseur, mémoire, écran, clavier, disques durs...) utilisés pour traiter les données. Dans ce contexte, l'ordinateur est un terme générique qui désigne un équipement informatique permettant de traiter des informations selon des séquences d'instructions (les programmes) qui constituent le logiciel. Le terme ORDINATEUR a été proposé par le philologue Jacques Perret en avril 1955 en réponse à une demande d'IBM France qui estimait le mot « calculateur » (*computer*) bien trop restrictif en regard des possibilités de ces machines (voir la proposition de Jacques Perret en annexe 1.6.1 page 31).

Définition 1.2 : MATÉRIEL
Le matériel informatique est un ensemble de dispositifs physiques utilisés pour traiter automatiquement des informations.

Définition 1.3 : LOGICIEL
Le logiciel est un ensemble structuré d'instructions décrivant un traitement d'informations à faire réaliser par un matériel informatique.

Un ordinateur n'est rien d'autre qu'une machine effectuant des opérations simples sur des séquences de signaux électriques, lesquels sont conditionnés de manière à ne pouvoir prendre

Fig. 1.1 : DÉFINITIONS DE L'ACADÉMIE (1)

INFORMATIQUE *n. f. et adj. XXe siècle. Dérivé d'information sur le modèle de mathématique, électronique. 1. N. f. Science du traitement rationnel et automatique de l'information ; l'ensemble des applications de cette science. 2. Adj. Qui se rapporte à l'informatique. Système informatique, ensemble des moyens qui permettent de conserver, de traiter et de transmettre l'information.*

INSTRUCTION *n. f. XIVe siècle. Emprunté du latin instructio, « action d'adapter, de ranger », puis « instruction ». Ordre, indication qu'on donne à quelqu'un pour la conduite d'une affaire ; directive, consigne. Le plus souvent au pluriel. Des instructions verbales, écrites. Donner des instructions à ses collaborateurs. Instructions pour la mise en marche d'un appareil. Par anal. INFORM. Consigne formulée dans un langage de programmation, selon un code.*

LOGICIEL *n. m. XXe siècle. Dérivé de logique. INFORM. Ensemble structuré de programmes remplissant une fonction déterminée, permettant l'accomplissement d'une tâche donnée.*

MATÉRIEL *adj. et n. XIIIe siècle. Emprunté du latin materialis, de même sens. INFORM. Ensemble des éléments physiques employés pour le traitement des données, par opposition aux logiciels.*

ORDINATEUR *n. m. XVe siècle, au sens de « celui qui institue quelque chose » ; XXe siècle, au sens actuel. Emprunté du latin ordinator, « celui qui règle, met en ordre ; ordonnateur ». Équipement informatique comprenant les organes nécessaires à son fonctionnement autonome, qui assure, en exécutant les instructions d'un ensemble structuré de programmes, le traitement rapide de données codées sous forme numérique qui peuvent être conservées et transmises.*

que deux états seulement (par exemple un potentiel électrique maximum ou minimum). Ces séquences de signaux obéissent à une logique binaire du type « tout ou rien » et peuvent donc être considérés conventionnellement comme des suites de nombres ne prenant jamais que les deux valeurs 0 et 1 : un ordinateur est donc incapable de traiter autre chose que des nombres binaires. Toute information d'un autre type doit être convertie, ou codée, en format binaire. C'est vrai non seulement pour les données que l'on souhaite traiter (les nombres, les textes, les images, les sons, les vidéos, etc.), mais aussi pour les programmes, c'est-à-dire les séquences d'instructions que l'on va fournir à la machine pour lui dire ce qu'elle doit faire avec ces données.

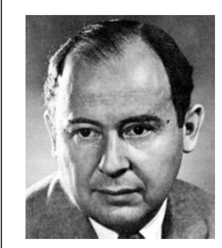
L'architecture matérielle d'un ordinateur repose sur le modèle de Von Neumann (figure 1.2) qui distingue classiquement 4 parties (figure 1.3) :

1. L'unité arithmétique et logique, ou unité de traitement, effectue les opérations de base.
2. L'unité de contrôle séquence les opérations.
3. La mémoire contient à la fois les données et le programme qui dira à l'unité de contrôle quels calculs faire sur ces données. La mémoire se divise entre mémoire vive volatile (programmes et données en cours de fonctionnement) et mémoire de masse permanente (programmes et données de base de la machine).
4. Les entrées-sorties sont des dispositifs permettant de communiquer avec le monde extérieur (écran, clavier, souris. . .).

Les 2 premières parties sont souvent rassemblées au sein d'une même structure : le microprocesseur (la « puce »), unité centrale de l'ordinateur.

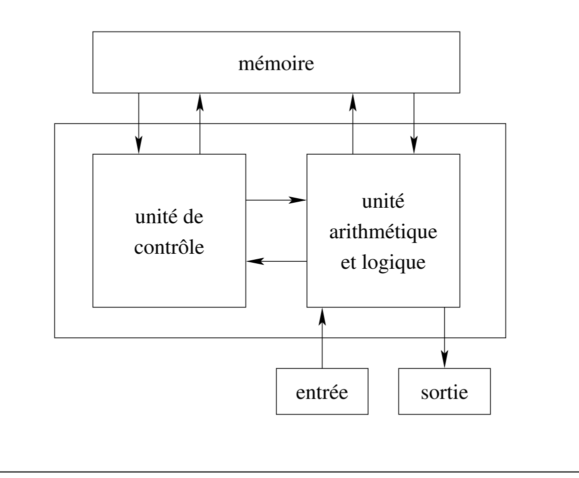
Dans ce cours, nous ne nous intéresserons qu'aux aspects logiciels de l'informatique.

Fig. 1.2 : John Von Neumann (1903-1957)



Mathématicien américain d'origine hongroise : il a donné son nom à l'architecture de von Neumann utilisée dans la quasi totalité des ordinateurs modernes (voir figure 1.3 ci-dessous).

Fig. 1.3 : Architecture de Von Neumann



1.1.2 Algorithmique

Exemple 1.1 : Mode d'emploi d'un télécopieur

Extrait du mode d'emploi d'un télécopieur concernant l'envoi d'un document.

1. *Insérez le document dans le chargeur automatique.*
2. *Composez le numéro de fax du destinataire à l'aide du pavé numérique.*
3. *Enfoncez la touche* envoi *pour lancer l'émission.*

Ce mode d'emploi précise comment envoyer un fax. Il est composé d'une suite ordonnée d'instructions (insérez. . ., composez. . ., enfoncez. . .) qui manipulent des données (document, chargeur

automatique, numéro de fax, pavé numérique, touche ENVOI) pour réaliser la tâche désirée (envoi d'un document).

Chacun a déjà été confronté à de tels documents pour faire fonctionner un appareil plus ou moins réticent et donc, consciemment ou non, a déjà exécuté un algorithme (ie. exécuter la suite d'instructions dans l'ordre annoncé, figure 1.4). ■TD1.1

TD 1.1 : Dessins sur la plage : exécution (1)
On cherche à faire dessiner une figure géométrique sur la plage à quelqu'un qui a les yeux bandés.
Quelle figure géométrique dessine-t-on en exécutant la suite d'instructions ci-dessous ?

1. *avance de 10 pas,*
2. *tourne à gauche d'un angle de 120°,*
3. *avance de 10 pas,*
4. *tourne à gauche d'un angle de 120°,*
5. *avance de 10 pas.*

Définition 1.4 : ALGORITHME
Un algorithme est une suite ordonnée d'instructions qui indique la démarche à suivre pour résoudre une série de problèmes équivalents.

Exemple 1.2 : Trouver son chemin
Extrait d'un dialogue entre un touriste égaré et un autochtone.
- *Pourriez-vous m'indiquer le chemin de la gare, s'il vous plait ?*
- *Oui bien sûr : vous allez tout droit jusqu'au prochain carrefour, vous prenez à gauche au carrefour et ensuite la troisième à droite, et vous verrez la gare juste en face de vous.*
- *Merci.*

Dans ce dialogue, la réponse de l'autochtone est la description d'une suite ordonnée d'instructions (allez tout droit, prenez à gauche, prenez la troisième à droite) qui manipulent des données (carrefour, rues) pour réaliser la tâche désirée (aller à la gare). Ici encore, chacun a déjà été confronté à ce genre de situation et donc, consciemment ou non, a déjà construit un algorithme dans sa tête (ie. définir la suite d'instructions pour réaliser une tâche). Mais quand on définit un algorithme, celui-ci ne doit contenir que des instructions compréhensibles par celui qui devra l'exécuter (des humains dans les 2 exemples précédents). ■TD1.2

TD 1.2 : Dessins sur la plage : conception (1)
Faire dessiner une spirale rectangulaire de 5 côtés, le plus petit côté faisant 2 pas de long et chaque côté fait un pas de plus que le précédent.

Dans ce cours, nous devrons apprendre à définir des algorithmes pour qu'ils soient compréhensibles — et donc exécutables — par un ordinateur.

Fig. 1.4 : Définitions de l'Académie (2)
ALGORITHME *n. m. XIIIe siècle, augorisme. Altération, sous l'influence du grec arithmos, « nombre », d'algorisme, qui, par l'espagnol, remonte à l'arabe Al-Khuwarizmi, surnom d'un mathématicien. Méthode de calcul qui indique la démarche à suivre pour résoudre une série de problèmes équivalents en appliquant dans un ordre précis une suite finie de règles.*
DONNÉE *n. f. XIIIe siècle, au sens de « distribution, aumône » ; XVIIIe siècle, comme terme de mathématiques. Participe passé féminin substantivé de donner au sens de « indiquer, dire ». 1. Fait ou principe indiscuté, ou considéré comme tel, sur lequel se fonde un raisonnement ; constatation servant de base à un examen, une recherche, une découverte.* INFORM. *Représentation d'une information sous une forme conventionnelle adaptée à son exploitation.*

Définition 1.5 : ALGORITHMIQUE
L'algorithmique est la science des algorithmes.

L'algorithmique s'intéresse à l'art de construire des algorithmes ainsi qu'à caractériser leur validité, leur robustesse, leur réutilisabilité, leur complexité ou leur efficacité.

Définition 1.6 : VALIDITÉ D'UN ALGORITHME
La validité d'un algorithme est son aptitude à réaliser exactement la tâche pour laquelle il a été conçu.

Si l'on reprend l'exemple 1.2 de l'algorithme de recherche du chemin de la gare, l'étude de sa validité consiste à s'assurer qu'on arrive effectivement à la gare en exécutant scrupuleusement les instructions dans l'ordre annoncé.

Définition 1.7 : ROBUSTESSE D'UN ALGORITHME
La robustesse d'un algorithme est son aptitude à se protéger de conditions anormales d'utilisation.

Dans l'exemple 1.2, la question de la robustesse de l'algorithme se pose par exemple si le chemin proposé a été pensé pour un piéton, alors que le « touriste égaré » est en voiture et que la « troisième à droite » est en sens interdit.

Définition 1.8 : RÉUTILISABILITÉ D'UN ALGORITHME
La réutilisabilité d'un algorithme est son aptitude à être réutilisé pour résoudre des tâches équivalentes à celle pour laquelle il a été conçu.

L'algorithme de recherche du chemin de la gare est-il réutilisable tel quel pour se rendre à la mairie ? A priori non, sauf si la mairie est juste à côté de la gare.

Définition 1.9 : COMPLEXITÉ D'UN ALGORITHME
La complexité d'un algorithme est le nombre d'instructions élémentaires à exécuter pour réaliser la tâche pour laquelle il a été conçu.

Si le « touriste égaré » est un piéton, la complexité de l'algorithme de recherche de chemin peut se compter en nombre de pas pour arriver à la gare.

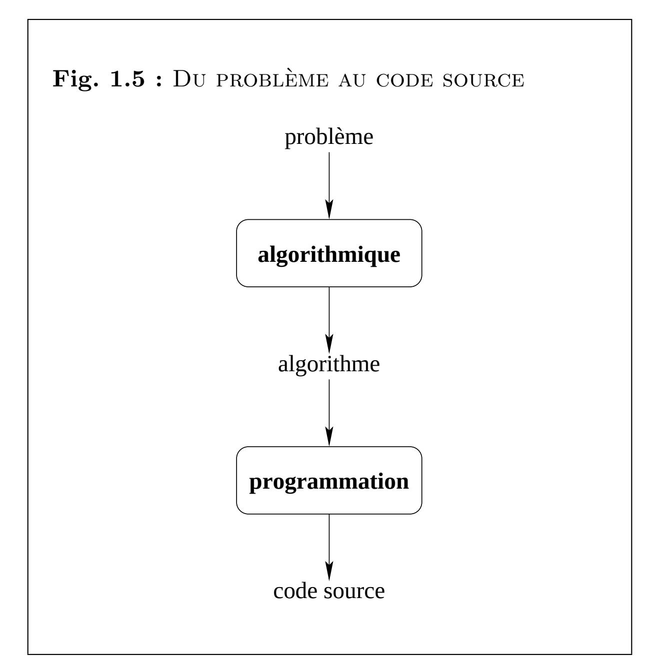
Définition 1.10 : EFFICACITÉ D'UN ALGORITHME
L'efficacité d'un algorithme est son aptitude à utiliser de manière optimale les ressources du matériel qui l'exécute.

N'existerait-il pas un raccourci piétonnier pour arriver plus vite à la gare ? ■TD1.3

L'algorithmique permet ainsi de passer d'un problème à résoudre à un algorithme qui décrit la démarche de résolution du problème. La programmation a alors pour rôle de traduire cet algorithme dans un langage « compréhensible » par l'ordinateur afin qu'il puisse exécuter l'algorithme automatiquement (figure 1.5).

TD 1.3 : PROPRIÉTÉS D'UN ALGORITHME
Reprendre le TD 1.1 et illustrer la validité, la robustesse, la réutilisabilité, la complexité et l'efficacité de l'algorithme proposé pour dessiner sur la plage.

Fig. 1.5 : DU PROBLÈME AU CODE SOURCE



1.1.3 Programmation

Un algorithme exprime la structure logique d'un programme informatique et de ce fait est indépendant du langage de programmation utilisé. Par contre, la traduction de l'algorithme dans un langage particulier dépend du langage choisi et sa mise en œuvre dépend également de la plateforme d'exécution.

La programmation d'un ordinateur consiste à lui « expliquer » en détail ce qu'il doit faire, en sachant qu'il ne « comprend » pas le langage humain, mais qu'il peut seulement effectuer un traitement automatique sur des séquences de 0 et de 1. Un programme n'est rien d'autre qu'une suite d'instructions, encodées en respectant de manière très stricte un ensemble de conventions fixées à l'avance par un langage informatique (figure 1.6). La machine décode alors ces instructions en associant à chaque « mot » du langage informatique une action précise. Le programme que nous écrivons dans le langage informatique à l'aide d'un éditeur (une sorte de traitement de texte spécialisé) est appelé programme source (ou code source).

Le seul « langage » que l'ordinateur puisse véritablement « comprendre » est donc très éloigné de ce que nous utilisons nous-mêmes. C'est une longue suite de 0 et de 1 (les « bits », *binary digit*) traités par groupes de 8 (les « octets », *byte*), 16, 32, ou même 64 (figure 1.6).

Définition 1.11 : BIT
Un bit est un chiffre binaire (0 ou 1). C'est l'unité élémentaire d'information.

Définition 1.12 : OCTET
Un octet est une unité d'information composée de 8 bits.

Exemple 1.3 : DESCRIPTION DE CLÉ USB
Sur le descriptif commercial d'une clé USB (figure 1.7), on peut lire :
- *Type de produit : Lecteur flash USB*
- *Taille du module : 512 Mo*
- *Type d'interface : Hi-Speed USB*

La « taille du module » est la capacité de stockage de la clé qui vaut ici 512 Mo (mégaoctets). Le préfixe « méga » est un multiplicateur décimal qui vaut 10^6 mais s'il est bien adapté au calcul décimal, il l'est moins au calcul binaire car il ne correspond pas à une puissance entière de 2. En effet, la puissance x de 2 telle que $2^x = 10^6$ vaut $x = 6 \cdot \dfrac{\log(10)}{\log(2)} \approx 19.93$. On choisit alors la

Fig. 1.6 : DÉFINITIONS DE L'ACADÉMIE (3)
LANGAGE. *n. m. XIIe siècle. Dérivé de langue. Système de signes, de symboles, élaboré à partir des langues naturelles et constituant un code qui les remplace dans certains cas déterminés (on parle aussi de langage symbolique). Le langage mathématique. Langage logique, fondé sur la logique formelle. Langage informatique. Langage de programmation.*
BIT *n. m. XXe siècle. Mot anglo-américain, contraction de binary digit, « chiffre binaire ». Chacun des deux chiffres, 0 et 1, de la numération binaire. En informatique, le bit est l'unité élémentaire d'information appelée aussi élément binaire.*
OCTET *n. m. XXe siècle. Dérivé savant du latin octo, « huit ». Unité d'information composée de huit bits.*

Fig. 1.7 : CLÉ USB (UNIVERSAL SERIAL BUS)



Dispositif matériel contenant une mémoire de masse (une mémoire flash ou un mini disque dur).

puissance entière de 2 immédiatement supérieure ($2^{20} = 1\,048\,576$) pour définir le Mo : 1 Mo = 1 048 576 octets. ■TD1.4

Pour « parler » à un ordinateur, il nous faudra donc utiliser des systèmes de traduction automatiques, capables de convertir en nombres binaires des suites de caractères formant des mots-clés (anglais en général) qui seront plus significatifs pour nous. Le système de traduction proprement dit s'appellera interpréteur ou bien compilateur, suivant la méthode utilisée pour effectuer la traduction (figure 1.8).

TD 1.4 : Unités d'information
Combien y a-t-il d'octets dans 1 ko (kilooctet), 1 Go (gigaoctet), 1 To (téraoctet), 1 Po (pétaoctet), 1 Eo (exaoctet), 1 Zo (zettaoctet) et 1 Yo (yottaoctet) ?

Définition 1.13 : compilateur
Un compilateur est un programme informatique qui traduit un langage, le langage source, en un autre, appelé le langage cible.

Définition 1.14 : interpréteur
Un interpréteur est un outil informatique (logiciel ou matériel) ayant pour tâche d'analyser et d'exécuter un programme écrit dans un langage source.

On appellera langage de programmation un ensemble de mots-clés (choisis arbitrairement) associé à un ensemble de règles très précises indiquant comment on peut assembler ces mots pour former des « phrases » que l'interpréteur ou le compilateur puisse traduire en langage machine (binaire). Un langage de programmation se distingue du langage mathématique par sa visée opérationnelle (ie. il doit être exécutable par une machine), de sorte qu'un langage de programmation est toujours un compromis entre sa puissance d'expression et sa possibilité d'exécution.

Définition 1.15 : langage de programmation
Un langage de programmation est un langage informatique, permettant à un humain d'écrire un code source qui sera analysé par un ordinateur.

Le code source subit ensuite une transformation ou une évaluation dans une forme exploitable par la machine, ce qui permet d'obtenir un programme. Les langages permettent souvent de faire abstraction des mécanismes bas niveaux de la machine, de sorte que le code source représentant une solution puisse être rédigé et compris par un humain.

Définition 1.16 : programmation
La programmation est l'activité de rédaction du code source d'un programme.

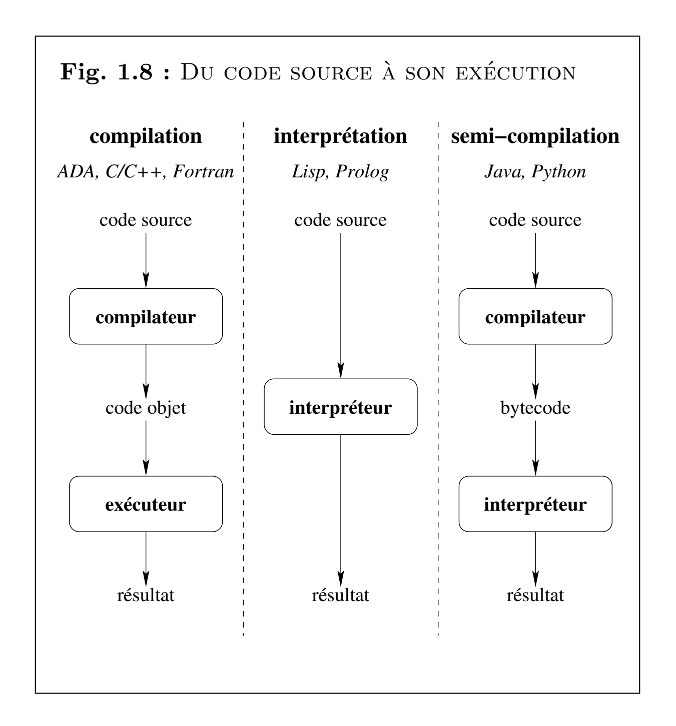


Fig. 1.8 : Du code source à son exécution

Compilation : La compilation consiste à traduire la totalité du code source en une fois. Le compilateur lit toutes les lignes du programme source et produit une nouvelle suite de codes appelé programme objet (ou code objet). Celui-ci peut désormais être exécuté indépendamment du compilateur et être conservé tel quel dans un fichier (« fichier exécutable »). Les langages Ada, C, C++ et Fortran sont des exemples de langages compilés.

Interprétation : L'interprétation consiste à traduire chaque ligne du programme source en quelques instructions du langage machine, qui sont ensuite directement exécutées au fur et à mesure (« au fil de l'eau »). Aucun programme objet n'est généré. L'interpréteur doit être utilisé chaque fois que l'on veut faire fonctionner le programme. Les langages Lisp et Prolog sont des exemples de langages interprétés.

L'interprétation est idéale lorsque l'on est en phase d'apprentissage du langage, ou en cours d'expérimentation sur un projet. Avec cette technique, on peut en effet tester immédiatement toute modification apportée au programme source, sans passer par une phase de compilation qui demande toujours du temps. Mais lorsqu'un projet comporte des fonctionnalités complexes qui doivent s'exécuter rapidement, la compilation est préférable : un programme compilé fonctionnera toujours plus vite que son homologue interprété, puisque dans cette technique l'ordinateur n'a plus à (re)traduire chaque instruction en code binaire avant qu'elle puisse être exécutée.

Semi-compilation : Certains langages tentent de combiner les deux techniques afin de retirer le meilleur de chacune. C'est le cas des langages Python et Java par exemple. De tels langages commencent par compiler le code source pour produire un code intermédiaire, similaire à un langage machine (mais pour une machine virtuelle), que l'on appelle bytecode, lequel sera ensuite transmis à un interpréteur pour l'exécution finale. Du point de vue de l'ordinateur, le bytecode est très facile à interpréter en langage machine. Cette interprétation sera donc beaucoup plus rapide que celle d'un code source.

Le fait de disposer en permanence d'un interpréteur permet de tester immédiatement n'importe quel petit morceau de programme. On pourra donc vérifier le bon fonctionnement de chaque composant d'une application au fur et à mesure de sa construction. L'interprétation du bytecode compilé n'est pas aussi rapide que celle d'un véritable code machine, mais elle est satisfaisante pour de très nombreux programmes.



Python : www.python.org

Dans ce cours, nous choisirons d'exprimer directement les algorithmes dans un langage informatique opérationnel : le langage Python [16], sans passer par un langage algorithmique intermédiaire.

1.2 Objectifs du cours

1.2.1 Objectifs thématiques

L'objectif principal des enseignements d'informatique S1 de l'ENIB est l'acquisition des notions fondamentales de l'algorithmique. Plus précisément, nous étudierons successivement :

1. les instructions de base permettant de décrire les algorithmes : affectation, tests, boucles ;
2. les procédures et les fonctions qui permettent de structurer et de réutiliser les algorithmes ; on parlera alors d'encapsulation, de préconditions, de portée des variables, de passage de paramètres, d'appels de fonctions, de récursivité et de jeux de tests ;
3. les structures de données linéaires : tableaux, listes, piles, files, qui améliorent la structuration des données manipulées par les algorithmes. A cette occasion, on évaluera la complexité et l'efficacité de certains algorithmes utilisant ces structures linéaires.

Ces différentes notions seront mise en œuvre à travers l'utilisation du langage Python. ■TD1.5

TD 1.5 : Première utilisation de Python
Se connecter sur un poste de travail d'une salle informatique.

1. *Lancer* Python.
2. *Utiliser* Python *comme une simple calculette.*
3. *Quitter* Python.

Ne pas oublier de se déconnecter du poste de travail.

1.2.2 Objectifs pédagogiques

Au cours du semestre S1, nous nous positionnerons principalement sur les 3 premiers niveaux de la taxonomie de Bloom qui constitue une référence pédagogique pour la classification des niveaux d'apprentissage (figure 1.9) : connaissance, compréhension, application. Les 3 derniers niveaux seront plutôt abordés au cours du semestre S2 (analyse, synthèse, évaluation).

1. Connaissance : mémorisation et restitution d'informations dans les mêmes termes.
2. Compréhension : restitution du sens des informations dans d'autres termes.
3. Application : utilisation de règles, principes ou algorithmes pour résoudre un problème, les règles n'étant pas fournies dans l'énoncé.
4. Analyse : identification des parties constituantes d'un tout pour en distinguer les idées.
5. Synthèse : réunion ou combinaison des parties pour former un tout.
6. Evaluation : formulation de jugements qualitatifs ou quantitatifs.

Afin de mieux nous situer par rapport aux différents types de pédagogie associés, nous « filerons » une métaphore musicale inspirée de [3].

Fig. 1.9 : Taxonomie de Bloom
Benjamin Bloom (1913-1999), psychologue américain spécialisé en pédagogie.

Connaître	: *définir, distinguer, acquérir, identifier, rappeler, reconnaître. . .*
Comprendre	: *traduire, illustrer, représenter, dire avec ses mots, distinguer, réécrire, réarranger, expliquer, démontrer. . .*
Appliquer	: *appliquer, généraliser, relier, choisir, développer, utiliser, employer, transférer, classer, restructurer. . .*
Analyser	: *distinguer, détecter, classer, reconnaître, catégoriser, déduire, discerner, comparer. . .*
Synthétiser	: *écrire, relater, produire, constituer, transmettre, modifier, créer, proposer, planifier, projeter, spécifier, combiner, classer, formuler. . .*
Evaluer	: *juger, argumenter, valider, décider, comparer. . .*

Remarque 1.1 : *L'annexe A page 207 présente quelques grands classiques « historiques ». On trouvera également une liste d'algorithmes classiques sur le site du National Institute of Standards and Technology* (NIST : www.nist.gov/dads).

Pédagogie par objectifs : Le solfège est l'étude des principes élémentaires de la musique et de sa notation : le musicien « fait ses gammes » et chaque exercice a un objectif précis pour évaluer l'apprentissage du « langage musical ». Il en va de même pour l'informaticien débutant confronté à l'apprentissage d'un langage algorithmique.

Pédagogie par l'exemple : L'apprentissage des grands classiques permet au musicien de s'approprier les bases du solfège en les appliquant à ces partitions connues et en les (re)jouant lui-même. L'informaticien débutant, en (re)codant lui-même des algorithmes bien connus, se constituera ainsi une base de réflexes de programmation en « imitant » ces algorithmes.

Pédagogie de l'erreur : Les bogues (*bugs*) sont à l'informaticien ce que les fausses notes sont aux musiciens : des erreurs. Ces erreurs sont nécessaires dans l'acquisition de la connaissance. Un élève a progressé si, après s'être trompé, il peut reconnaître qu'il s'est trompé, dire où et pourquoi il s'est trompé, et comment il recommencerait sans produire les mêmes erreurs.

Pédagogie par problèmes : Connaissant « ses » classiques et les bases du solfège, le musicien devenu plus autonome peut envisager sereinement la création de ses propres morceaux. Le développement d'un projet informatique ambitieux sera « mis en musique » au semestre S2.

Dans ce cours, nous adopterons ces différentes stratégies pédagogiques sans oublier qu'en informatique on apprend toujours mieux en faisant par soi-même.

1.2.3 Objectifs comportementaux

Nous cherchons à développer trois « qualités » comportementales chez l'informaticien débutant : la rigueur, la persévérance et l'autonomie.

Rigueur : Un ordinateur est une machine qui exécute vite et bien les instructions qu'on lui a « apprises ». Mais elle ne sait pas interpréter autre chose : même mineure, une erreur provoque le dysfonctionnement de la machine.

Exemple 1.4 : Erreur de syntaxe en langage C
Pour un « ; » oublié dans un programme C*, le code source n'est pas compilable et le compilateur nous avertit par ce type de message :*

```
fichier.c : In function ‘main’ :
fichier.c :7 : error : syntax error before "printf"
```

Même si un humain peut transiger sur le « ; », la machine ne le peut pas : l'ordinateur ne se contente pas de l'« à peu près ». La respect des consignes, la précision et l'exactitude sont donc de rigueur en informatique ! ■TD1.6

Persévérance : Face à l'intransigeance de la machine, le débutant est confronté à ses nombreuses erreurs (les siennes, pas celles de la machine !) et sa tendance naturelle est de passer à autre chose. Mais le papillonnage (ou *zapping*) est une très mauvaise stratégie en informatique : pour s'exécuter correctement, un programme doit être finalisé. L'informatique nécessite d'« aller au bout des choses ». ■TD1.7

Autonomie : Programmer soi-même les algorithmes qu'on a définis est sans doute le meilleur moyen pour mieux assimiler les principales structures algorithmiques et pour mieux comprendre ses erreurs en se confrontant à l'intransigeante impartialité de l'ordinateur, véritable « juge de paix » des informaticiens. Par ailleurs, l'évolution continuelle et soutenue des langages de programmation et des ordinateurs nécessitent de se tenir à jour régulièrement et seule l'autoformation systématique permet de « ne pas perdre pied ». ■TD1.8

Pratique personnelle et autoformation constituent ainsi deux piliers de l'autonomisation nécessaire de l'apprenti informaticien.

TD 1.6 : Erreur de syntaxe en Python
On considère la session Python *suivante :*

```
>>> x = 3
>>>  y = x
  File "<stdin>", line 1
    y = x
    ^
SyntaxError : invalid syntax
>>>
```

De quelle erreur de syntaxe s'agit-il ?

TD 1.7 : Dessins sur la plage : persévérance
Finir l'algorithme suivant qui cherche à dessiner un losange sur la plage.

1. *avance de 10 pas,*
2. *tourne à gauche d'un angle de* $60°$*,*

⋮

TD 1.8 : Autonomie
Trouver les définitions du mot autonomie et de son contraire (de son antonyme).

1.3 Organisation du cours

1.3.1 Présentiel

Les enseignements d'informatique S1 de l'ENIB sont dispensés lors de 42h de séances de cours-TD et de séances de laboratoire.

– Les cours-TD ont lieu 1 semaine sur 2 à raison de 3h par semaine, soit 21h de cours-TD sur toute la durée du semestre. Ils se déroulent dans une salle banalisée.
– Les séances de laboratoire ont lieu 1 semaine sur 2 en alternance avec les cours-TD à raison de 3h par semaine, soit 21h de laboratoire dans le semestre. Elles se déroulent en salle informatique.

Remarque 1.2 : *Un semestre s'étale sur 14 semaines d'enseignements, 7 semaines consacrées au cours et 7 semaines aux TD.*

1.3.2 Documents

Les principaux documents accompagnant les cours sont de 2 types : les supports de cours et les notes de cours.

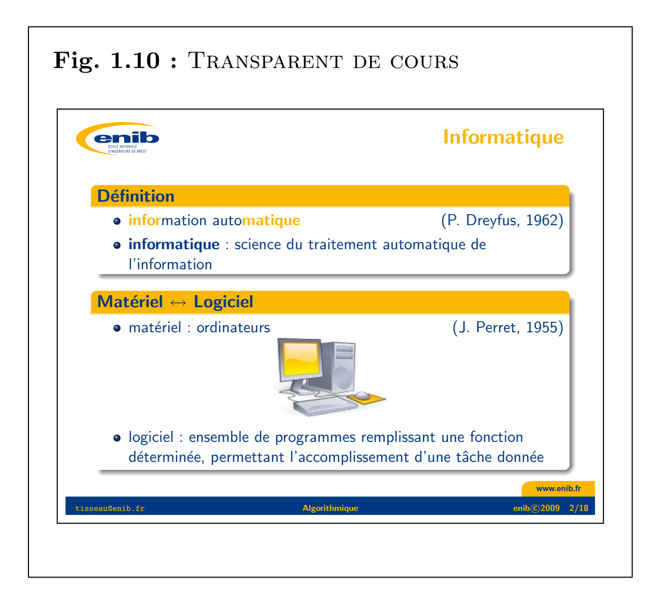


Fig. 1.10 : TRANSPARENT DE COURS

Support de cours : il s'agit de la copie papier des transparents projetés en présentiel (figure 1.10).

Notes de cours : il s'agit de notes qui complètent et précisent certains points présentés en cours. Ces notes proposent également les exercices de travaux dirigés qui sont étudiés en cours et au laboratoire. Le document que vous êtes en train de lire constitue le premier chapitre des notes du cours d'informatique S1 de l'ENIB. Il y en aura 3 autres sur les sujets suivants :

1. les instructions de base (chapitre 2 page 39),
2. les procédures et les fonctions (chapitre 3 page 99),
3. les structures de données linéaires (chapitre 4 page 157).

Un site WEB permet de retrouver ces documents au format `pdf` (*Portable Document Format*) ainsi que d'autres documents tels que le planning prévisionnel des cours et des contrôles, des exemples corrigés d'évaluation, les notes aux différents contrôles ou encore des liens vers des sites pertinents pour le cours. Un forum de discussions professeurs↔élèves est également ouvert sur ce site. ■TD1.9

TD 1.9 : SITE WEB D'INFORMATIQUE S1
Se connecter sur le site WEB *du cours d'informatique S1 de l'ENIB et vérifier que ces notes de cours sont bien disponibles sur le site au format* `pdf`.

La consultation régulière du site est indispensable pour se tenir au courant des dernières évolutions du cours : en cas d'ambiguïté, ce sont les informations de ce site qui feront foi.

1.3.3 Evaluations des apprentissages

L'évaluation des connaissances et des compétences acquises par les étudiants repose sur 4 types de contrôle : les contrôles d'attention, les contrôles de TD, les contrôles d'autoformation et les contrôles de compétences.

Contrôle d'attention : il s'agit d'un QCM (questionnaire à choix multiples) auquel il faut répondre individuellement sans document, en 5' en fin de cours, et dont les questions portent sur des points abordés pendant ce cours. Ce type de contrôle teste directement l'acquisition de connaissances au sens du « connaître » de la classification de Bloom (section 1.2.2 page 11). Il permet d'évaluer « à chaud » la capacité d'attention des étudiants et les incite à une présence attentive afin de bénéficier au maximum des heures de présence aux cours. ■TD1.10

TD 1.10 : EXEMPLE DE CONTRÔLE D'ATTENTION (1)
Répondre de mémoire aux questions suivantes (ie. sans rechercher les solutions dans les pages précédentes).

1. *Quels sont les 3 principaux thèmes informatiques abordés ?*
2. *Quelles sont les 4 principales stratégies pédagogiques suivies ?*
3. *Quelles sont les 3 principales qualités comportementales recherchées ?*

Des exemples de QCM sont systématiquement proposés dans les notes de cours comme par exemple celui du TD 1.19 page 20 pour cette introduction générale.

Contrôle de TD : il s'agit ici d'inciter les étudiants à préparer activement les séances de laboratoire. En début de chaque séance de laboratoire, chaque binôme doit répondre sans document en 10' aux questions d'un exercice de TD. L'exercice est choisi aléatoirement parmi les exercices de TD situés en marge des notes de cours et se rapportant au thème du TD. ■TD1.11

Il concerne le « comprendre » et l'« appliquer » de la taxonomie de Bloom (section 1.2.2 page 11).

Contrôle d'autoformation : un contrôle d'autoformation porte sur un thème prévu à l'avance et que l'étudiant doit approfondir par lui-même. Les thèmes retenus pour l'autoformation en S1 sont par exemple le calcul booléen, le codage des nombres ou encore la recherche d'éléments dans un tableau de données. ■TD1.12

Ces contrôles se déroulent pendant les séances de cours : ce sont des écrits individuels de 30' sans document qui concernent le « connaître », le « comprendre » et l'« appliquer » de la taxonomie de Bloom.

Contrôle de compétences : les contrôles de compétences (ou DS) durent 80' pendant une séance de cours. Plus longs, ils permettent d'aborder l'un des 3 derniers niveaux de la classification de Bloom (analyser, synthétiser, évaluer) comme le font les exercices de la section 1.5.4 page 23. ■TD1.13

Quel que soit le type de contrôle, un exercice cherche à évaluer un objectif particulier. Aussi, la notation exprimera la distance qui reste à parcourir pour atteindre cet objectif (figure 1.11) :

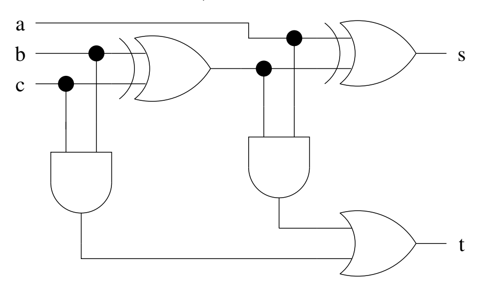
0 : « en plein dans le mille ! »	→ l'objectif est atteint
1 : « pas mal ! »	→ on se rapproche de l'objectif
2 : « juste au bord de la cible ! »	→ on est encore loin de l'objectif
3 : « la cible n'est pas touchée ! »	→ l'objectif n'est pas atteint

Ainsi, et pour changer de point de vue sur la notation, le contrôle est réussi lorsqu'on a 0 ! Il n'y a pas non plus de 1/2 point ou de 1/4 de point : le seul barême possible ne comporte que 4 niveaux : 0, 1, 2 et 3. On ne cherche donc pas à « grappiller » des points :

- on peut avoir 0 (objectif atteint) et avoir fait une ou deux erreurs bénignes en regard de l'objectif recherché ;
- on peut avoir 3 (objectif non atteint) et avoir quelques éléments de réponse corrects mais sans grand rapport avec l'objectif.

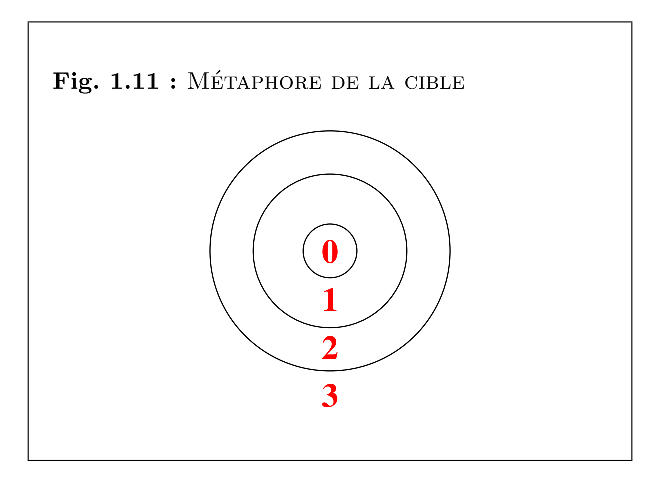
TD 1.11 : Exemple de contrôle de TD
Répondre aux questions du TD 1.10 situé dans la marge de la page 14.

TD 1.12 : Exemple de contrôle d'autoformation (1)
Etablir la table de vérité du circuit logique ci-dessous où a, b et c sont les entrées, s et t les sorties.



TD 1.13 : Exemple de contrôle des compétences
Répondre aux questions du TD 1.28 de la page 25.

Fig. 1.11 : Métaphore de la cible



Remarque 1.3 : *Une absence à un contrôle conduit à la note 4 (« la cible n'est pas visée »).*

1.3.4 Evaluation des enseignements

En fin de semestre, les étudiants organisent de manière anonyme et dans le respect des personnes, une évaluation individuelle des enseignements. Elle est structurée en 2 parties : un questionnaire de 10 à 20 questions (voir un exemple en annexe 1.6.2 page 33) auxquelles il faut répondre selon une grille pré-définie et une partie « expression libre » que l'étudiant remplit, ou non, à son gré.

L'ensemble des fiches d'évaluation est remis à l'équipe pédagogique d'Informatique S1, qui après en avoir pris connaissance, organise une rencontre spécifique avec les étudiants. Cette évaluation a pour objectif l'amélioration de la qualité des enseignements et de la pédagogie à partir de la perception qu'en ont les étudiants.

1.4 Méthodes de travail

Remarque 1.4 : *Ce document comporte 259 pages structurées en 4 chapitres, 3 annexes, 4 index et une bibliographie. Il propose 47 définitions, 86 figures, 39 exemples, 79 remarques, 128 exercices et 5 contrôles types corrigés. En moyenne, au cours des 14 semaines que dure le cours d'informatique S1 de l'ENIB, le travail personnel hebdomadaire consiste donc à lire entre 15 et 20 pages de ce cours en retenant 3 à 4 définitions et en faisant entre 7 et 10 exercices.*

TD 1.14 : Nombre de contrôles
Après consultation du calendrier prévisionnel de l'annexe 1.6.3 page 34, donner le nombre et le type de contrôles prévus au calendrier du semestre.

Il ne s'agit pas ici d'imposer des méthodes de travail, mais de fournir des pistes pour ceux qui en cherchent. Dans tous les cas, l'expérience montre que :

1. la seule présence, même attentive et active, aux séances de cours et de laboratoire ne suffit pas : il faut prévoir un temps de travail personnel qui, selon l'étudiant et la matière, peut aller de 50% à 150% du temps de présence en cours ;
2. la régularité dans le travail personnel est un gage d'apprentissage plus efficace.

Le calendrier prévisionnel des enseignements et des contrôles associés est distribué en début de semestre (un exemple de planning est donné en annexe 1.6.3 page 34). ■TD1.14
Les documents de cours sont distribués au moins 15 jours avant les séances correspondantes (sauf en ce qui concerne les 2 premières semaines de cours évidemment). Par ailleurs, à tout moment, le calendrier et les documents sont disponibles sur le site Web d'Informatique S1.

1.4.1 Avant, pendant et après le cours

Préparation : Certains cours débutent par un contrôle d'autoformation (voir section 1.3.3) dont le thème est en rapport avec certains aspects du cours qui suivra immédiatement. Il est donc nécessaire d'avoir étudié avant et par soi-même le sujet du contrôle.

En général, on trouvera les informations nécessaires soit directement sur le site d'Informatique S1, soit dans les références bibliographiques données en fin des notes de cours (voir

par exemple les références de ce chapitre en page 271), soit dans les polycopiés d'autres cours de l'ENIB (mathématiques, électronique, productique, microprocesseurs...), soit encore sur internet en s'assurant de la qualité du site (préférer des sites universitaires ou d'écoles dont l'activité principale est d'enseigner ces matières).

Par exemple, il est nécessaire d'avoir assimilé les principes du calcul booléen pour maîtriser les tests et les boucles du langage algorithmique. C'est pourquoi, une autoformation est imposée sur ce domaine déjà connu des étudiants. Un contrôle-type d'autoformation sur le calcul booléen pourrait par exemple être composé des TD 1.12 page 15 et 1.15 ci-contre. ■ TD1.15

Pour chaque contrôle d'autoformation, un exemple corrigé est disponible sur le site d'Informatique S1. Il est donc fortement recommandé de travailler ce contrôle-type : après avoir revu par soi-même les principales notions du domaine, faire le contrôle sans consulter au préalable la solution proposée.

Participation : Par respect pour les autres participants, la ponctualité est de rigueur pour l'étudiant comme pour le professeur. Mais assister à un cours n'a d'intérêt que dans le cas d'une présence attentive et soutenue : le temps passé en cours pour assimiler les nouvelles notions sera gagné sur le temps de travail personnel futur nécessaire à leur assimilation.

Chaque page des supports de cours est constituée d'une copie d'un transparent de cours dans la demi-page supérieure alors que la demi-page inférieure est volontairement laissée vierge pour que l'étudiant prenne des notes pendant le cours (figure 1.12).

La prise de note systématique est en effet un facteur qui favorise l'attention. Un contrôle de type QCM en fin de cours évaluera l'attention des étudiants (voir section 1.3.3). ■ TD1.16

Le cours est illustré par des exemples et des exercices : à ces occasions, c'est une participation active de l'étudiant qui est attendue dans une démarche volontaire d'assimilation du cours « à chaud ».

Appropriation : Dans la mesure du possible, il faut relire ses propres notes ainsi que les notes de cours le soir même afin de « fixer » les principales idées du cours. La révision proprement dite peut venir ultérieurement quelques jours plus tard (et non quelques semaines ni quelques mois après).

Les définitions, comme toute définition, sont à apprendre par cœur . Une technique possible est de lire 2 fois de suite à voix haute la définition en détachant distinctement les différentes expressions (exemple : « l'informatique » « est la science » « du traitement » « automatique » « de l'information »), puis l'écrire de mémoire.

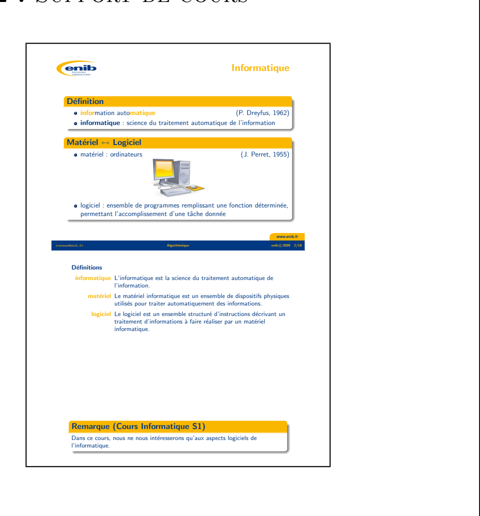
TD 1.15 : Exemple de contrôle d'autoformation (2)
Exprimer en la développant la négation des expressions booléennes suivantes :

1. `(0 < x) and (x < 3)`
2. `(x < -2) or (x > 4)`
3. `a and (not b)`
4. `(not a) or b`

TD 1.16 : Exemple de contrôle d'attention (2)
Répondre de mémoire aux questions suivantes (ie. sans rechercher les solutions dans les pages précédentes).

1. *Quels sont les 4 types de contrôle proposés ?*
2. *Quels sont les documents que l'on peut trouver sur le site Web du cours ?*

Fig. 1.12 : Support de cours



Pour réviser le cours, faire systématiquement les TD au moment où ils sont référencés dans les notes par le symbole ■ . C'est particulièrement vrai avec les exemples pour lesquels sont associés des exercices en marge des notes de cours (exemple : l'exemple 1.1 de la page 5 et le TD 1.1 associé en marge de la même page). Lorsque l'exemple est un algorithme, il faut systématiquement se mettre mentalement à la place de la machine qui va les exécuter (on parle alors d'« empathie numérique ») afin de vérifier le résultat obtenu. Pour cela, il faut être méthodique et rigoureux. Et petit à petit, à force de pratique, l'expérience fera qu'on « verra » le résultat produit par les instructions au fur et à mesure qu'on les écrit. Naturellement, cet apprentissage est long, et demande des heures de travail patient. Aussi, dans un premier temps, il faut éviter de sauter les étapes : la vérification méthodique, pas à pas, de chacun des algorithmes représente plus de la moitié du travail à accomplir [5].

1.4.2 Avant, pendant et après le laboratoire

Préparation : Faire les exercices situés dans les marges de ces notes de cours est non seulement une façon de réviser son cours (voir section 1.4.1) mais aussi de préparer les séances de laboratoire. Pour ces notes de cours, la liste complète des exercices est donnée en annexe B page 223. Un de ces exercices choisi aléatoirement fera l'objet d'une évaluation en début de chaque séance de TD (voir section 1.3.3). ■TD1.17

Tous ces exercices ne nécessitent pas une longue phase de réflexion comme les TD 2.7 ou 1.13 (→ 1.28). Certains au contraire ne présentent aucune difficulté particulière comme les TD 1.14 et 1.17 qui demandent simplement de compter les contrôles et les exercices. D'autres comme les TD 1.10 et 1.16 font appel à la mémoire, d'autres encore à la recherche d'informations comme les TD 1.4 et 1.8, ou à une mise en œuvre pratique comme les TD 1.5 et 1.9.

TD 1.17 : Nombres d'exercices de TD
Combien d'exercices y avait-il à faire avant celui-ci ?

Participation : Ici encore, par respect pour les autres participants, la ponctualité est de rigueur pour l'étudiant comme pour le professeur.

En informatique, lorsqu'on travaille en binôme, il faut régulièrement alterner les rôles entre l'« écrivain » qui manipule clavier et souris et le « lecteur » qui vérifie la production de l'écrivain. A la fin du semestre, chaque étudiant doit être devenu un « lecteur-écrivain ». La pratique est en effet nécessaire pour « apprivoiser » l'environnement de travail afin que la machine devienne « transparente » et que seul subsiste le problème algorithmique à résoudre. ■TD1.18

TD 1.18 : Environnement de travail
Sur un poste de travail d'une salle informatique :

1. *Quel est le type de clavier ?*
2. *Comment ouvre-t-on un terminal ?*
3. *Comment lance-t-on* Python *?*
4. *Où sont stockés les fichiers de travail ?*

Pour certains exercices, la solution est donnée dans les notes de cours (en section 1.5.5 page 26 pour ce chapitre). Pendant les séances de laboratoire, il faut donc savoir « jouer le jeu » pour ne pas simplement « recopier » la solution proposée (il existe d'ailleurs d'autres solutions pour la plupart des exercices).

Un apprenti programmeur est toujours confronté à ses propres erreurs (revoir le TD 1.6 page 13 par exemple). En général, ce sont des erreurs simples à corriger à condition de lire les messages d'erreur et de faire l'effort de les comprendre avant d'appeler le professeur « à l'aide ».

Exemple 1.5 : Erreur de nom en Python

Voici une erreur classique d'une variable non correctement initialisée en Python *:*

```
>>> print(x)
Traceback (most recent call last) :
  File "<stdin>", line 1, in ?
NameError : name 'x' is not defined
>>>
```

En Python, la dernière ligne du message d'erreur est la plus « parlante » au débutant.

Appropriation : Avant le cours suivant, il faut refaire les TD qui ont posé des problèmes et faire les exercices qui n'ont pas pu être abordés en séance de laboratoire (les solutions de ces exercices complémentaires sont données dans les notes de cours).

1.4.3 Apprendre en faisant

1. En bas de la page 17, il est dit :

   « Les définitions, comme toute définition, sont à apprendre par cœur. »

   Connaissez-vous les 16 définitions introduites dans ce chapitre ? Elles sont clairement identifiables grâce au mot-clé « **Définition** ».
2. En haut de la page 18, il est dit :

   « Pour réviser le cours, faire systématiquement les TD au moment où ils sont référencés dans les notes par le symbole ■. »

   Avez-vous cherché à résoudre les 18 exercices de TD proposés jusqu'à présent ? Ils sont clairement identifiables grâce au mot-clé « **TD** » situé dans la marge.

Il n'y a pas de miracle, c'est votre travail personnel qui est le meilleur gage de vos apprentissages. On apprend toujours mieux en faisant par soi-même.

Remarque 1.5 : *La dernière phrase de cette section a déjà été écrite dans le texte qui précède. A propos de quoi ? En quelle page ?*

1.5 Exercices complémentaires

1.5.1 Connaître

TD 1.19 : QCM (1)

(un seul item correct par question)

Remarque 1.6 : *Parmi les 4 items de la question ci-contre, un seul item définit l'informatique, les 3 autres définissent d'autres sciences. Lesquelles ?*

1. *L'informatique est la science*
   (a) *des dispositifs dont le fonctionnement dépend de la circulation d'électrons*
   (b) *des signaux électriques porteurs d'information ou d'énergie*
   (c) *du traitement automatique de l'information*
   (d) *de la commande des appareils fonctionnant sans intervention humaine*
2. *Le logiciel est*
   (a) *la mémoire de l'ordinateur*
   (b) *le traitement automatique de l'information*
   (c) *l'ensemble des données manipulées par les instructions*
   (d) *un ensemble structuré d'instructions décrivant un traitement d'informations à faire réaliser par un matériel informatique*
3. *L'algorithmique est la science*
   (a) *du traitement automatique de l'information*
   (b) *des algorithmes*
   (c) *des langages de programmation*
   (d) *des instructions*
4. *Un algorithme est*
   (a) *un ensemble de programmes remplissant une fonction déterminée, permettant l'accomplissement d'une tâche donnée*
   (b) *une suite ordonnée d'instructions qui indique la démarche à suivre pour résoudre une série de problèmes équivalents*
   (c) *le nombre d'instructions élémentaires à exécuter pour réaliser une tâche donnée*
   (d) *un ensemble de dispositifs physiques utilisés pour traiter automatiquement des informations*

5. *La validité d'un algorithme est son aptitude*
   (a) *à utiliser de manière optimale les ressources du matériel qui l'exécute*
   (b) *à se protéger de conditions anormales d'utilisation*
   (c) *à calculer le nombre d'instructions élémentaires nécessaires pour réaliser la tâche pour laquelle il a été conçu*
   (d) *à réaliser exactement la tâche pour laquelle il a été conçu*
6. *La complexité d'un algorithme est*
   (a) *le nombre de fois où l'algorithme est utilisé dans un programme*
   (b) *le nombre de données manipulées par les instructions de l'algorithme*
   (c) *le nombre d'octets occupés en mémoire par l'algorithme*
   (d) *le nombre d'instructions élémentaires à exécuter pour réaliser la tâche pour laquelle il a été conçu*
7. *Un bit est*
   (a) *un chiffre binaire*
   (b) *composé de 8 chiffres binaires*
   (c) *un chiffre héxadécimal*
   (d) *un mot d'un langage informatique*
8. *Un compilateur*
   (a) *exécute le code source*
   (b) *exécute le bytecode*
   (c) *traduit un code source en code objet*
   (d) *exécute le code objet*

Remarque 1.7 : *Parmi les 4 items de la question ci-contre, un seul item définit la validité d'un algorithme, les 3 autres se rapportent à d'autres propriétés des algorithmes. Lesquelles ?*

TD 1.20 : Puissance de calcul

Donner l'ordre de grandeur en instructions par seconde des machines suivantes (voir [6] par exemple) :

1. *le premier micro-ordinateur de type PC,*
2. *une console de jeu actuelle,*

3. *un micro-ordinateur actuel,*
4. Deeper-Blue *: l'ordinateur qui a « battu » Kasparov aux échecs en 1997,*
5. *le plus puissant ordinateur actuel.*

TD 1.21 : Stockage de données
Donner l'ordre de grandeur en octets pour stocker en mémoire (voir [6] par exemple) :

Remarque 1.8 : *TD 1.21 : voir aussi TD 1.4*

1. *une page d'un livre,*
2. *une encyclopédie en 20 volumes,*
3. *une photo couleur,*
4. *une heure de vidéo,*
5. *une minute de son,*
6. *une heure de son.*

1.5.2 Comprendre

TD 1.22 : Dessins sur la plage : exécution (2)

Remarque 1.9 : *TD 1.22 : voir aussi TD 1.1*

1. *Quelle figure géométrique dessine-t-on en exécutant la suite d'instructions ci-dessous ?*
   (a) *avance de 3 pas,*
   (b) *tourne à gauche d'un angle de* 90°*,*
   (c) *avance de 4 pas,*
   (d) *rejoindre le point de départ.*
2. *Combien de pas a-t-on fait au total pour rejoindre le point de départ ?*

TD 1.23 : Dessins sur la plage : conception (2)
Reprendre le TD 1.2 et illustrer la validité, la robustesse, la réutilisabilité, la complexité et l'efficacité de l'algorithme proposé pour dessiner une spirale rectangulaire.

Remarque 1.10 : *TD 1.23 : voir aussi TD 1.2*

1.5.3 Appliquer

Remarque 1.11 : *TD 1.24 : voir aussi TD 1.7*

TD 1.24 : Tracés de polygones réguliers
On cherche à faire dessiner une figure polygonale (figure 1.13) sur la plage à quelqu'un qui a les yeux bandés. Pour cela, on ne dispose que de 2 commandes orales : avancer de n pas en avant (n est un nombre entier de pas) et tourner à gauche d'un angle θ (rotation sur place de θ).

1. *Faire dessiner un pentagone régulier de 10 pas de côté.*
2. *Faire dessiner un hexagone régulier de 10 pas de côté.*
3. *Faire dessiner un octogone régulier de 10 pas de côté.*
4. *Faire dessiner un polygone régulier de n côtés de 10 pas chacun.*

1.5.4 Analyser

TD 1.25 : La multiplication « à la russe »

La technique de multiplication dite « à la russe » consiste à diviser par 2 le multiplicateur (et ensuite les quotients obtenus), jusqu'à un quotient nul, à noter les restes, et à multiplier parallèlement le multiplicande par 2. On additionne alors les multiples obtenus du multiplicande correspondant aux restes non nuls.

Exemple : 68×123 $(= 8364)$

multiplicande $M \times 2$	*multiplicateur* $m \div 2$	*reste* $m \bmod 2$	*somme partielle*
123	*68*	*0*	$(0 \times 123) + 0$
246	*34*	*0*	$(0 \times 246) + 0$
492	*17*	*1*	$(1 \times 492) + 0$
984	*8*	*0*	$(0 \times 984) + 492$
1968	*4*	*0*	$(0 \times 1968) + 492$
3936	*2*	*0*	$(0 \times 3936) + 492$
7872	*1*	*1*	$(1 \times 7872) + 492$
		$68 \times 123 =$	*8364*

Effectuer les multiplications suivantes selon la technique « à la russe ».

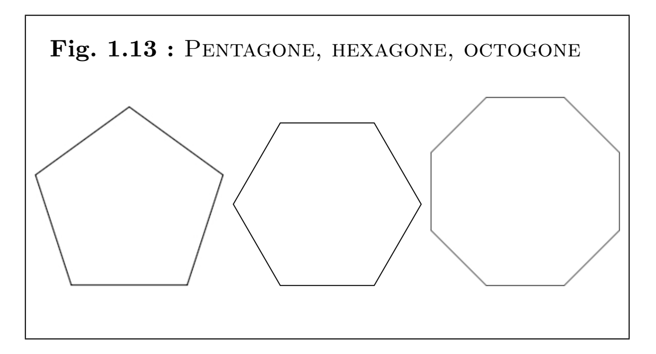
1. 64×96 $(= 6144)$
2. 45×239 $(= 10755)$

TD 1.26 : La multiplication arabe

On considère ici le texte d'Ibn al-Banna concernant la multiplication à l'aide de tableaux [2].

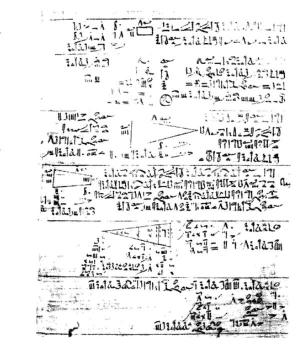
« Tu construis un quadrilatère que tu subdivises verticalement et horizontalement en autant de bandes qu'il y a de positions dans les deux nombres multipliés. Tu divises diagonalement les carrés obtenus, à l'aide de diagonales allant du coin inférieur gauche au coin supérieur droit (figure 1.14).

Fig. 1.13 : Pentagone, hexagone, octogone



Remarque 1.12 : *La multiplication en Egypte antique.*

La multiplication « à la russe » est une variante connue d'une technique égyptienne décrite dans le papyrus Rhind (environ -1650). Le scribe Ahmès y expose des problèmes de géométrie et d'arithmétique (qui viennent en partie des Babyloniens) dont cette technique de multiplication.



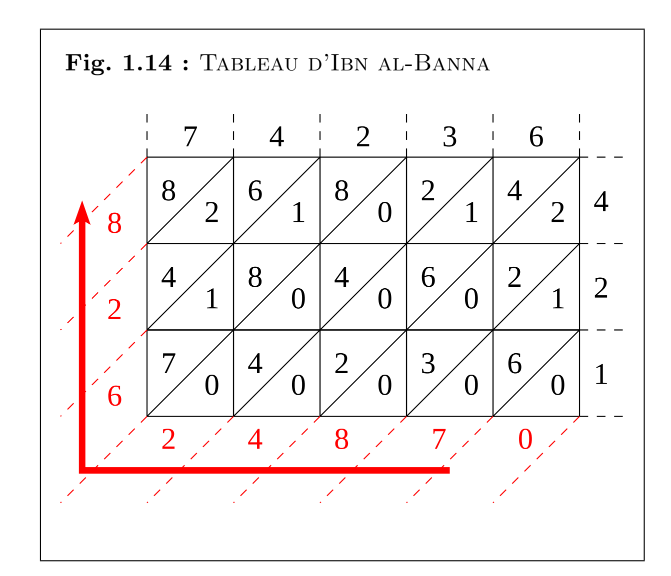


Fig. 1.14 : Tableau d'Ibn al-Banna

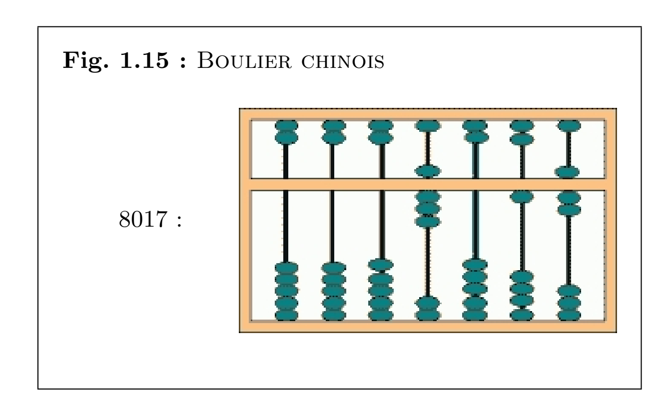


Fig. 1.15 : Boulier chinois

Tu places le multiplicande au-dessus du quadrilatère, en faisant correspondre chacune de ses positions à une colonne[1]. Puis, tu places le multiplicateur à gauche ou à droite du quadrilatère, de telle sorte qu'il descende avec lui en faisant correspondre également chacune de ses positions à une ligne[2]. Puis, tu multiplies, l'une après l'autre, chacune des positions du multiplicande du carré par toutes les positions du multiplicateur, et tu poses le résultat partiel correspondant à chaque position dans le carré où se coupent respectivement leur colonne et leur ligne, en plaçant les unités au-dessus de la diagonale et les dizaines en dessous. Puis, tu commences à additionner, en partant du coin supérieur gauche : tu additionnes ce qui est entre les diagonales, sans effacer, en plaçant chaque nombre dans sa position, en transférant les dizaines de chaque somme partielle à la diagonale suivante et en les ajoutant à ce qui y figure.

La somme que tu obtiendras sera le résultat. »

En utilisant la méthode du tableau d'Ibn al-Banna, calculer 63247×124 *(*$= 7842628$*).*

TD 1.27 : La division chinoise

Dans sa version actuelle, le boulier chinois se compose d'un nombre variable de tringles serties dans un cadre rectangulaire [2]. Sur chacune de ces tringles, deux étages de boules séparées par une barre transversale peuvent coulisser librement (figure 1.15). La notation des nombres repose sur le principe de la numération de position : chacune des 2 boules du haut vaut 5 unités et chacune des 5 boules du bas vaut 1 unité. Seules comptent les boules situées dans la région transversale.

Il existe des règles spéciales de division pour chaque diviseur de 1 à 9. On considère ici les 7 règles de division par 7 (figure 1.16) :

1. « qi-yi xia jia san » *: 7-1 ? ajouter 3 en dessous !*
2. « qi-er xia jia liu » *: 7-2 ? ajouter 6 en dessous !*
3. « qi-san si sheng er » *: 7-3 ? 4 reste 2 !*
4. « qi-si wu sheng wu » *: 7-4 ? 5 reste 5 !*
5. « qi-wu qi sheng yi » *: 7-5 ? 7 reste 1 !*
6. « qi-liu ba sheng si » *: 7-6 ? 8 reste 4 !*
7. « feng-qi jin yi » *: 7-7 ? 1 monté !*

Ces règles ne sont pas des règles logiques, mais de simples procédés mnémotechniques indiquant ce qu'il convient de faire selon la situation. Leur énoncé débute par le rappel du diviseur, ici 7, et se poursuit par l'énoncé du dividende, par exemple 3 : 7-3. Le reste de la règle indique

---

[1] L'écriture du nombre s'effectue de droite à gauche (exemple : 352 s'écrira donc 253).

[2] L'écriture du nombre s'effectue de bas en haut (exemple : $\begin{smallmatrix}3\\5\\2\end{smallmatrix}$ s'écrira donc $\begin{smallmatrix}2\\5\\3\end{smallmatrix}$).

quelles manipulations effectuées, ajouts ou retraits de boules. Il faut également savoir que le dividende étant posé sur le boulier, on doit appliquer les règles aux chiffres successifs du dividende, en commençant par celui dont l'ordre est le plus élevé. « ajouter en dessous » veut dire « mettre des boules au rang immédiatement inférieur (à droite) au rang considéré » et « monter » veut dire « mettre des boules au rang immédiatement supérieur (à gauche) au rang considéré ».

Pour effectuer la division d'un nombre par 7, on pose le dividende à droite sur le boulier et le diviseur (7) à gauche. On opère sur les chiffres successifs du dividende en commençant par celui d'ordre le plus élevé (le plus à gauche). Les règles précédemment énoncées sont appliquées systématiquement.

Utiliser un boulier chinois pour diviser 1234 par 7 ($1234 = 176 \times 7 + 2$).

$$\begin{array}{ccccc} 1 & 0 & 0 & 0 & 0 \\ \hline 2 & 1 & 2 & 3 & 4 \end{array} \rightarrow \begin{array}{ccccc} 1 & 0 & 1 & 1 & 0 \\ \hline 2 & 1 & 2 & 1 & 2 \end{array}$$

Fig. 1.16 : Règles de la division par 7

Règle	Avant				Après			
7-1	1	0	0	0	1	0	0	0
	2	0	1	0	2	0	1	3
7-2	1	0	0	0	1	0	0	1
	2	0	2	0	2	0	2	1
7-3	1	0	0	0	1	0	0	0
	2	0	3	0	2	0	4	2
7-4	1	0	0	0	1	0	1	1
	2	0	4	0	2	0	0	0
7-5	1	0	1	0	1	0	1	0
	2	0	0	0	2	0	2	1
7-6	1	0	1	0	1	0	1	0
	2	0	1	0	2	0	3	4
7-7	1	0	1	0	1	0	0	0
	2	0	2	0	2	1	0	0

TD 1.28 : Le calcul Shadok

Les cerveaux des Shadoks avaient une capacité tout à fait limitée [15]. Ils ne comportaient en tout que 4 cases. Comme ils n'avaient que 4 cases, évidemment les Shadoks ne connaissaient pas plus de 4 mots : ga, bu, zo et meu *(figure 1.17). Etant donné qu'avec 4 mots, ils ne pouvaient pas compter plus loin que 4, le Professeur Shadoko avait réformé tout ça :*

- *Quand il n'y a pas de Shadok, on dit* ga *et on écrit* ga.
- *Quand il y a un Shadok de plus, on dit* bu *et on écrit* bu.
- *Quand il y a encore un Shadok, on dit* zo *et on écrit* zo.
- *Et quand il y en a encore un autre, on dit* meu *et on écrit* meu.

Tout le monde applaudissait très fort et trouvait ça génial sauf le Devin Plombier qui disait qu'on n'avait pas idée d'inculquer à des enfants des bêtises pareilles et que Shadoko, il fallait le condamner. Il fut très applaudi aussi. Les mathématiques, cela les intéressait, bien sûr, mais brûler le professeur, c'était intéressant aussi, faut dire. Il fut décidé à l'unanimité qu'on le laisserait parler et qu'on le brûlerait après, à la récréation.

- *Répétez avec moi :* meu zo bu ga. . .ga bu zo meu.
- *Et après ! ricanait le Plombier.*
- *Si je mets un Shadok en plus, évidemment, je n'ai plus assez de mots pour les compter, alors c'est très simple : on les jette dans une poubelle, et je dis que j'ai* bu *poubelle. Et pour ne pas confondre avec le* bu *du début, je dis qu'il n'y a pas de Shadok à côté de la poubelle et j'écris* bu ga. bu *Shadok à côté de la poubelle :* bu bu. *Un autre :* bu zo. *Encore un autre :* bu meu. *On continue.* zo *poubelles et pas de Shadok à côté :* zo ga. . .meu

poubelles et MEU *Shadoks à côté :* MEU MEU*. Arrivé là, si je mets un Shadok en plus, il me faut une autre poubelle. Mais comme je n'ai plus de mots pour compter les poubelles, je m'en débarrasse en les jetant dans une grande poubelle. J'écris* BU *grande poubelle avec pas de petite poubelle et pas de Shadok à côté :* BU GA GA*, et on continue. . .*BU GA BU, BU GA ZO*. . .*MEU MEU ZO, MEU MEU MEU*. Quand on arrive là et qu'on a trop de grandes poubelles pour pouvoir les compter, eh bien, on les met dans une super-poubelle, on écrit* BU GA GA GA*, et on continue. . .(figure 1.18).*


Fig. 1.17 : Les Shadoks : ga bu zo meu

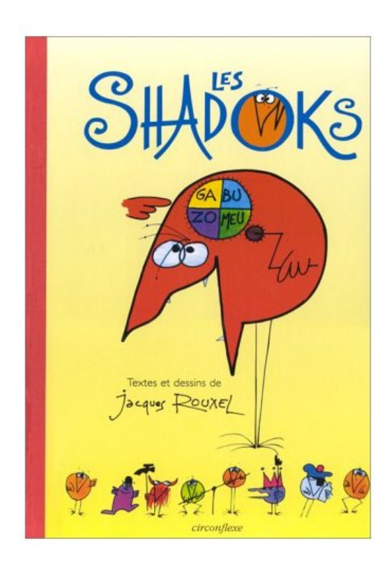


Fig. 1.18 : Les 18 premiers nombres Shadok

0 : GA	*6* : BU ZO	*12* : MEU GA
1 : BU	*7* : BU MEU	*13* : MEU BU
2 : ZO	*8* : ZO GA	*14* : MEU ZO
3 : MEU	*9* : ZO BU	*15* : MEU MEU
4 : BU GA	*10* : ZO ZO	*16* : BU GA GA
5 : BU BU	*11* : ZO MEU	*17* : BU GA BU

1. *Quels sont les entiers décimaux représentés en « base Shadok » par les expressions suivantes ?*
   (a) GA GA
   (b) BU BU BU
   (c) ZO ZO ZO ZO
   (d) MEU MEU MEU MEU MEU
2. *Effectuer les calculs Shadok suivants.*
   (a) ZO ZO MEU + BU GA MEU
   (b) MEU GA MEU − BU MEU GA
   (c) ZO MEU MEU × BU GA MEU
   (d) ZO ZO ZO MEU ÷ BU GA ZO

1.5.5 Solutions des exercices

TD 1.19 : QCM (1).
Les bonnes réponses sont extraites directement du texte de la section 1.1 :
1c, 2d, 3b, 4b, 5d, 6d, 7a, 8c

TD 1.20 : Puissance de calcul.
L'unité de puissance est donné ici en Mips (« million d'instructions par seconde »).

1. le premier micro-ordinateur de type PC : $\approx 10^{-1}$ Mips
2. une console de jeu actuelle : $\approx 10^{4}$ Mips
3. un micro-ordinateur actuel : $\approx 10^{4}$ Mips
4. *Deeper-Blue* : l'ordinateur qui a « battu » Kasparov aux échecs en 1997 : $\approx 10^{7}$ Mips

5. le plus puissant ordinateur actuel : $\approx 10^9$ Mips

TD 1.21 : Stockage de données.

1. une page d'un livre : $\approx$ 50 lignes de 80 caractères = 4 ko
2. une encyclopédie en 20 volumes : $\approx$ 20 volumes de 1000 pages de 50 lignes de 80 caractères = 80 Mo (sans images !)
3. une photo couleur : $\approx$ de 1Mo à 100 Mo selon la qualité
4. une heure de vidéo : $\approx$ de 500 Mo à 2 Go selon la qualité vidéo (DivX, MPEG-2...)
5. une minute de son : $\approx$ de 1 Mo (MP3) à 10 Mo selon la qualité du son
6. une heure de son : $\approx$ de 60 Mo à 600 Mo selon la qualité du son

TD 1.22 : Dessins sur la plage : exécution.

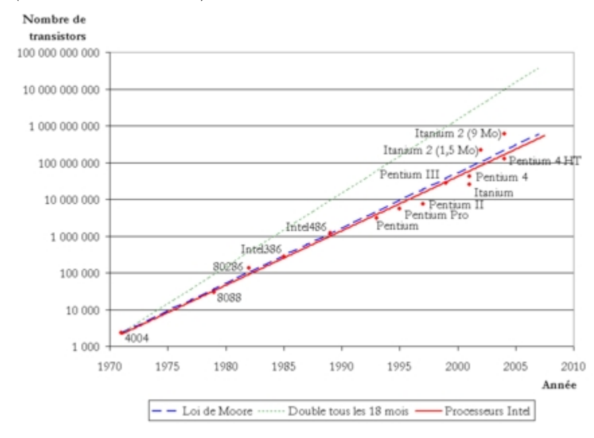
1. On trace un triangle rectangle dont l'hypothénuse fait 5 pas de long (d'après le théorème de Pythagore : $5^2 = 3^2 + 4^2$).
2. On a donc marché $3 + 4 + 5 = 12$ pas.

TD 1.23 : Dessins sur la plage : conception.
Imaginons l'algorithme de tracé suivant :

1. avance de 2 pas,
2. tourne à gauche de 90°,
3. avance de 3 pas,
4. tourne à gauche de 90°,
5. avance de 4 pas,
6. tourne à gauche de 90°,
7. avance de 5 pas,
8. tourne à gauche de 90°,
9. avance de 6 pas.

– validité : on doit au moins vérifier que la figure obtenue à toutes les caractéristiques recherchées : spirale rectangulaire de 5 côtés, le plus petit côté faisant 2 pas de long et chaque côté fait un pas de plus que le précédent (figure 1.19).

Remarque 1.13 : *Loi de Moore.*
Gordon Earl Moore est le co-fondateur avec Robert Noyce et Andrew Grove de la société Intel en 1968 (fabriquant n°1 mondial de microprocesseurs). En 1965, il expliquait que la complexité des semiconducteurs doublait tous les dix-huit mois à coût constant depuis 1959, date de leur invention. En 1975, il précise sa « première loi » en affirmant que le nombre de transistors des microprocesseurs sur une puce de silicium double tous les deux ans (« deuxième loi »).



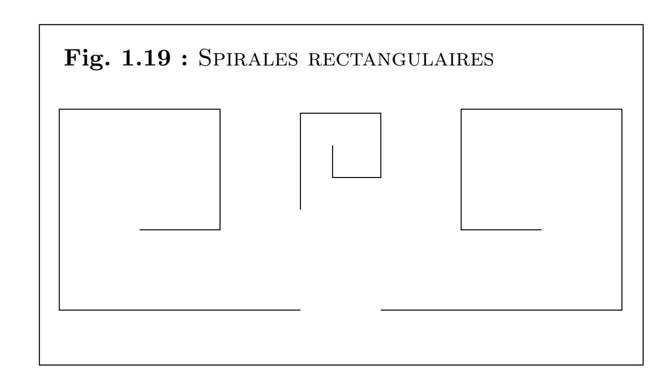


Fig. 1.19 : Spirales rectangulaires

- robustesse : cet algorithme suppose qu'on a suffisamment de place pour tracer une spirale (le dernier côté fait 6 pas de long) ; s'il fonctionne correctement sur une plage, il ne fonctionnera certainement plus dans un placard.
- réutilisabilité : il existe une infinité de spirales rectangulaires qui ont les caractéristiques attendues ; il suffit de penser à changer l'orientation initiale ou la longueur du pas par exemple. On ne pourra donc pas utiliser l'algorithme tel quel dans toutes les configurations : il aurait fallu paramétrer l'angle de rotation et la longueur du pas.
- complexité : on peut la caractériser par le nombre de pas : $2+3+4+5+6=20$ pas.
- efficacité : si la complexité se calcule en nombre de pas comme ci-dessus, on pourrait imaginer par exemple que la fréquence des pas soit plus grande (« fréquence d'horloge ») ou que 5 personnes prennent en charge chacun un côté de la spirale pour gagner du temps (« système multi-processeurs »).

TD 1.24 : Tracés de polygones réguliers.

1. Pentagone régulier de 10 pas de côté.
   (a) avance de 10 pas,
   (b) tourne à gauche de $(360/5)°$,
   (c) avance de 10 pas,
   (d) tourne à gauche de $(360/5)°$,
   (e) avance de 10 pas,
   (f) tourne à gauche de $(360/5)°$,
   (g) avance de 10 pas,
   (h) tourne à gauche de $(360/5)°$,
   (i) avance de 10 pas,
   (j) tourne à gauche de $(360/5)°$.

   On remarque qu'on a effectué 5 fois de suite les 2 instructions suivantes :
   (a) avance de 10 pas,
   (b) tourne à gauche de $(360/5)°$.

   Pour simplifier, on écrira plutôt :
   Répète 5 fois de suite les 2 instructions
   (a) avance de 10 pas,
   (b) tourne à gauche de $(360/5)°$.

   C'est ce que nous ferons dans les exemples suivants.
2. Hexagone régulier de 10 pas de côté.
   Répète 6 fois de suite les 2 instructions
   (a) avance de 10 pas,
   (b) tourne à gauche de $(360/6)°$,
3. Octogone régulier de 10 pas de côté.
   Répète 8 fois de suite les 2 instructions

   (a) avance de 10 pas,
   (b) tourne à gauche de $(360/8)^\circ$,

4. Polygone régulier de n côtés de 10 pas chacun.
   Répète n fois de suite les 2 instructions
   (a) avance de 10 pas,
   (b) tourne à gauche de $(360/n)^\circ$,

TD 1.25 : Multiplication « à la russe ».

1.

multiplicande $M \times 2$	multiplicateur $m \div 2$	reste m mod 2	somme partielle
96	64	0	$(0 \times 96) + 0$
192	32	0	$(0 \times 192) + 0$
384	16	0	$(1 \times 384) + 0$
768	8	0	$(0 \times 768) + 0$
1536	4	0	$(0 \times 1536) + 0$
3072	2	0	$(0 \times 3072) + 0$
6144	1	1	$(1 \times 6144) + 0$
		$64 \times 96 =$	6144

2.

multiplicande $M \times 2$	multiplicateur $m \div 2$	reste m mod 2	somme partielle
239	45	1	$(1 \times 239) + 0$
478	22	0	$(0 \times 478) + 239$
956	11	1	$(1 \times 956) + 239$
1912	5	1	$(1 \times 1912) + 1195$
3824	2	0	$(0 \times 3824) + 3107$
7648	1	1	$(1 \times 7648) + 3107$
		$45 \times 239 =$	10755

TD 1.26 : Multiplication arabe : $63247 \times 124 = 7842628$.

Remarque 1.14 : *Boulier chinois : division par 3.*

1. « san-yi sanshi-yi » *: trois-un ? trente et un !*
   (on pose 3 à la place du 1, et on ajoute 1 à droite)
2. « san-er liushi-er » *: trois-deux ? soixante deux !*
   (on pose 6 à la place du 2, et on ajoute 2 à droite)
3. « feng san jin yi-shi » *: trois-trois ? dizaine montée !*
   (on pose 0 à la place du 3, et on ajoute 1 à gauche)

Effectuer la division $2271 \div 3$ *avec le boulier chinois.*

Remarque 1.15 : *Un entier positif en base* b *est représenté par une suite de chiffres* $(r_n r_{n-1} \ldots r_1 r_0)_b$ *où les* r_i *sont des chiffres de la base* b $(0 \leq r_i < b)$*. Ce nombre a pour valeur :*

$$r_n b^n + r_{n-1} b^{n-1} + \ldots + r_1 b^1 + r_0 b^0 = \sum_{i=0}^{i=n} r_i b^i$$

Un nombre fractionnaire (nombre avec des chiffres après la virgule : $(r_n r_{n-1} \ldots r_1 r_0 . r_{-1} r_{-2} \ldots)_b$*) est défini sur un sous-ensemble borné, incomplet et fini des rationnels. Un tel nombre a pour valeur :*

$$r_n b^n + r_{n-1} b^{n-1} + \ldots + r_0 b^0 + r_{-1} b^{-1} + r_{-2} b^{-2} + \ldots$$

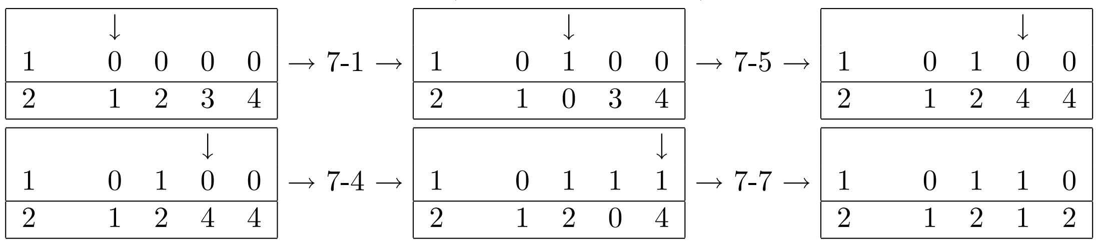
En pratique, le nombre de chiffres après la virgule est limité par la taille physique en machine.

$$(r_n r_{n-1} \ldots r_1 r_0 . r_{-1} r_{-2} \ldots r_{-m})_b = \sum_{i=-m}^{i=n} r_i b^i$$

	7	4	2	3	6	
8	8 / 2	6 / 1	8	2 / 1	4 / 2	4
2	4 / 1	8	4	6	2 / 1	2
6	7	4	2	3	6	1
	2	4	8	7		

résultat

TD 1.27 : Division chinoise : $1234 \div 7$ $(1234 = 176 \times 7 + 2)$.



		↓			
1		0	0	0	0
2		1	2	3	4

→ 7-1 →

			↓		
1		0	1	0	0
2		1	0	3	4

→ 7-5 →

				↓	
1		0	1	0	0
2		1	2	4	4

				↓	
1		0	1	0	0
2		1	2	4	4

→ 7-4 →

					↓
1		0	1	1	1
2		1	2	0	4

→ 7-7 →

1		0	1	1	0
2		1	2	1	2

TD 1.28 : Calcul Shadok.

Le système Shadok est un système de numération en base 4 :
GA = 0, BU = 1, ZO = 2 et MEU = 3.

1. (a) GA GA $= (00)_4 = 0$
   (b) BU BU BU $= (111)_4 = 1 \cdot 4^2 + 1 \cdot 4^1 + 1 \cdot 4^0 = 16 + 4 + 1 = 21$
   (c) ZO ZO ZO ZO $= (2222)_4 = 2 \cdot 4^3 + 2 \cdot 4^2 + 2 \cdot 4^1 + 2 \cdot 4^0 = 128 + 32 + 8 + 2 = 170$
   (d) MEU MEU MEU MEU MEU $= (33333)_4 = 3 \cdot 4^4 + 3 \cdot 4^3 + 3 \cdot 4^2 + 3 \cdot 4^1 + 3 \cdot 4^0 = 768 + 192 + 48 + 12 + 3 = 1023$
2. (a) ZO ZO MEU + BU GA MEU $= (223)_4 + (103)_4 = (332)_4 = 43 + 19 = 62$
   (b) MEU GA MEU − BU MEU GA $= (303)_4 - (130)_4 = (113)_4 = 51 - 28 = 23$
   (c) ZO MEU MEU × BU GA MEU $= (233)_4 \times (103)_4 = (31331)_4 = 47 \times 19 = 893$
   (d) ZO ZO ZO MEU ÷ BU GA ZO $= (2223)_4 \div (102)_4 = (21)_4 = 171 \div 18 = 9$

1.6 Annexes

1.6.1 Lettre de Jacques Perret

Au printemps de 1955, IBM France s'apprêtait à construire dans ses ateliers de Corbeil-Essonnes (consacrés jusque-là au montage des machines mécanographiques — tabulatrices, trieuses, ...— de technologie électromécanique) les premières machines électroniques destinées au traitement de l'information. Aux États-Unis ces nouvelles machines étaient désignées sous le vocable Electronic Data Processing System. Le mot « computer » était plutôt réservé aux machines scientifiques et se traduisait aisément en « calculateur » ou « calculatrice ». Sollicité par la direction de l'usine de Corbeil-Essonnes, François Girard, alors responsable du service promotion générale publicité, décida de consulter un de ses anciens maîtres, Jacques Perret, professeur de philologie latine à la Sorbonne. A cet effet il écrit une lettre à la signature de C. de Waldner, président d'IBM France. Il décrit sommairement la nature et les fonctions des nouvelles machines. Il accompagne sa lettre de brochures illustrant les machines mécanographiques. Le 16 avril, le professeur Perret lui répond. L'ordinateur IBM 650 peut commencer sa carrière. Protégé pendant quelques mois par IBM France, le mot fut rapidement adopté par un public de spécialistes, de chefs d'entreprises et par l'administration. IBM décida de le laisser dans le domaine public (d'après le site de la 10ème semaine de la langue française et de la francophonie www.semainelf.culture.fr/site2005/dixmots).

Le 16 IV 1955

Cher Monsieur,

Que diriez-vous d'« ordinateur » ? C'est un mot correctement formé, qui se trouve même dans le Littré comme adjectif désignant Dieu qui met de l'ordre dans le monde. Un mot de ce genre a l'avantage de donner aisément un verbe « ordiner », un nom d'action « ordination ». L'inconvénient est que « ordination » désigne une cérémonie religieuse ; mais les deux champs de signification (religion et comptabilité) sont si éloignés et la cérémonie d'ordination connue, je crois, de si peu de personnes que l'inconvénient est peut-être mineur. D'ailleurs votre machine serait « ordinateur » (et non ordination) et ce mot est tout à fait sorti de l'usage théologique. « Systémateur » serait un néologisme, mais qui ne me paraît pas offensant ; il permet « systématisé » ; - mais « système » ne me semble guère utilisable - « combinateur » a l'inconvénient du sens péjoratif de « combine » ; « combiner » est usuel donc peu capable de devenir technique ; « combination » ne me paraît guère viable à cause de la proximité de « combinaison ». Mais les Allemands ont bien leurs « combinats » (sorte de trusts, je crois), si bien

Fig. 1.20 : Le premier ordinateur (1946) *ENIAC (Electronic Numerical Integrator Analyser and Computer).*

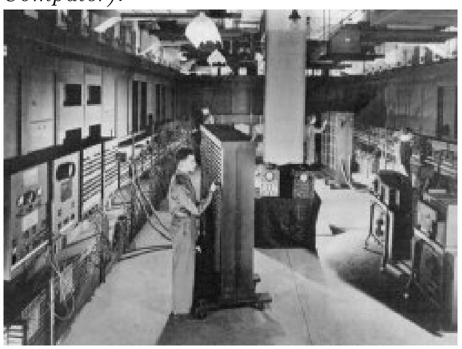


Fig. 1.21 : Premiers micro-ordinateurs



Apple II (1977)



ZX 81 (1981)



IBM PC (1981)



Macintosh (1984)

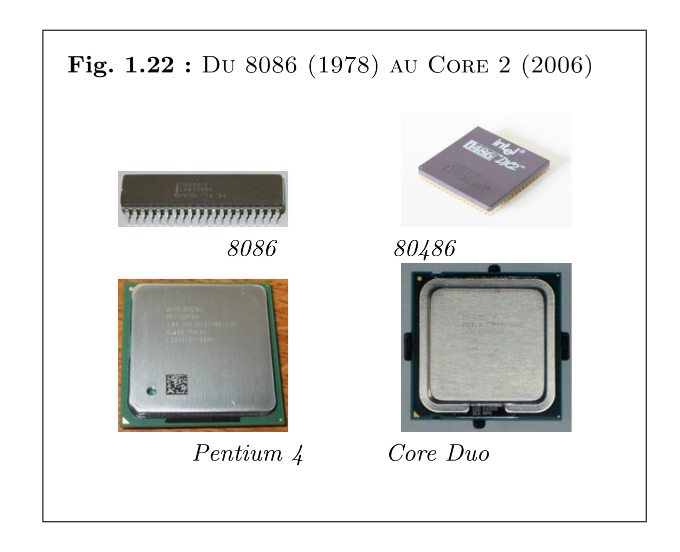
que le mot aurait peut-être des possibilités autres que celles qu'évoque « combine ».

« Congesteur », « digesteur » évoquent trop « congestion » et « digestion ». « Synthétiseur » ne me paraît pas un mot assez neuf pour designer un objet spécifique, déterminé comme votre machine. En relisant les brochures que vous m'avez données, je vois que plusieurs de vos appareils sont désignés par des noms d'agent féminins (trieuse, tabulatrice). « Ordinatrice » serait parfaitement possible et aurait même l'avantage de séparer plus encore votre machine du vocabulaire de la théologie. Il y a possibilité aussi d'ajouter à un nom d'agent un complément : « ordinatrice d'éléments complexes » ou un élément de composition, par ex. : « sélecto-systémateur ». - « Sélecto- ordinateur » a l'inconvénient de 2 « o » en hiatus, comme « électro-ordinatrice ». Il me semble que je pencherais pour « ordinatrice électronique ». Je souhaite que ces suggestions stimulent, orientent vos propres facultés d'invention. N'hésitez pas à me donner un coup de téléphone si vous avez une idée qui vous paraisse requérir l'avis d'un philologue.

Vôtre.

J. Perret

Fig. 1.22 : Du 8086 (1978) au Core 2 (2006)



8086 *80486*

Pentium 4 *Core Duo*

Fig. 1.23 : Micro-ordinateurs récents

PDA *Tablette*

iPhone *Wii*

1.6.2 Exemple de questionnaire d'évaluation

Proposition	**1**	**2**	**3**	**4**	×
Vos connaissances antérieures étaient suffisantes pour suivre ce cours	☐	☐	☐	☐	☐
Les objectifs du cours ont été clairement définis	☐	☐	☐	☐	☐
Vous êtes satisfait des supports de cours fournis	☐	☐	☐	☐	☐
Vous êtes satisfait de l'équilibre cours/TD/TP	☐	☐	☐	☐	☐
Vous êtes satisfait de la présentation de ce cours (clarté d'expression...)	☐	☐	☐	☐	☐
Si vous avez plusieurs professeurs, la cohérence de l'enseignement vous a semblé assurée	☐	☐	☐	☐	☐
Le rythme de travail vous a permis de suivre et de comprendre	☐	☐	☐	☐	☐
Le temps alloué à cette matière vous a semblé satisfaisant	☐	☐	☐	☐	☐
Cette matière a nécessité beaucoup de travail personnel	☐	☐	☐	☐	☐
Vous avez eu l'impression de progresser	☐	☐	☐	☐	☐
Les contrôles sont adaptés à l'objectif et au niveau du cours	☐	☐	☐	☐	☐
Vous êtes satisfait du mode d'évaluation	☐	☐	☐	☐	☐
Après les contrôles, les enseignants fournissent des commentaires qui aident à mieux maitriser la matière	☐	☐	☐	☐	☐
Vous êtes satisfait des conditions de travail (salles de cours, matériel utilisé...)	☐	☐	☐	☐	☐
Respect du programme pédagogique	☐	☐	☐	☐	☐
1 : Très satisfait , **2 :**Satisfait , **3 :**Peu satisfait , **4 :**Pas satisfait ; × **:**Ne me concerne pas					

1.6.3 Exemple de planning

Les enseignements d'Informatique S1 s'étalent sur 14 semaines.

Semaine	**Cours-TD**	**TD**
1	introduction générale instructions de base	—
2	—	environnement de programmation **CTD1** : affectation et tests instructions de base
3	**CAF1** : calculs booléens instructions de base	—
4	—	**CTD2** : boucles simples instructions de base
5	instructions de base **DS1** : instructions de base	—
6	—	**CTD3** : boucles imbriquées instructions de base
7	procédures et fonctions	—
8	—	**CTD4** : spécification de fonctions procédures et fonctions
9	**CAF2** : codage des nombres procédures et fonctions	—
10	—	**CTD5** : implémentation de fonctions procédures et fonctions
11	procédures et fonctions	—
12	—	**CTD6** : appel de fonctions procédures et fonctions
13	**DS2** : procédures et fonctions structures linéaires	—
14	—	**CTD7** : manipulation de listes structures linéaires

CAF : contrôle d'autoformation (écrit individuel, 30', sans document)
CTD : contrôle de travaux dirigés (sur machine par binôme, 10', sans document)
DS : devoir surveillé (écrit individuel, 80', sans document)

1.6.4 Informatique à l'ENIB

ENIB : www.enib.fr



Enseignements de premier cycle

Les 4 semestres du premier cycle (S1 à S4) sont communs à tous les élèves de l'ENIB. Les enseignements d'informatique sont précisés dans le tableau ci-dessous.

Semestre	Thème	Horaires
S1	Algorithmique	42 h
S2	Méthode de développement	42 h
S3	Programmation procédurale	42 h
S3	Programmation par objets	42 h
S4	Programmation pour l'embarqué	42 h

Enseignements du cycle ingénieur

Les 2 premiers semestres du cycle ingénieur (S5 et S6) sont communs à tous les élèves de l'ENIB.

Semestre	Thème	Horaires
S5	Programmation par objets	42 h
S6	Programmation par objets	42 h
S6	Modèles pour l'ingénierie des systèmes	42 h
S6	Bases de données	21 h

Au cours du 3ème semestre du cycle ingénieur (S7), les élèves choisissent une option parmi trois : électronique, informatique ou mécatronique. Les modules de l'option informatique sont listés dans le tableau ci-dessous. Le 4ème semestre (S8) est consacré à un Projet Professionnalisant en Equipe (PPE) ou à un stage en entreprise.

Semestre	Thème	Horaires
S7	Systèmes embarqués 1	42 h
S7	Systèmes embarqués 2	42 h
S7	Réseaux	42 h
S7	Administration Systèmes et Réseaux	42 h
S7	Génie Logiciel	42 h
S7	Systèmes d'Information	42 h
S7	Interfaces Homme-Machine	42 h
S7	Applications Réparties	42 h

Au cours du 5ème semestre du cycle ingénieur (S9), les étudiants doivent suivre 9 modules scientifiques, les modules à caractère informatique sont donnés dans le tableau ci-dessous. Le dernier semestre (S10) est consacré au stage en entreprise.

Semestre	Thème	Horaires
S9	Systèmes distribués	42 h
S9	Contrôle adaptatif	42 h
S9	Styles de programmation	42 h
S9	Intelligence artificielle et Simulation	42 h
S9	Réalité Virtuelle	42 h

Recherches en informatique

L'ENIB accueille trois laboratoires de recherche correspondant aux trois options : électronique, informatique et mécatronique.

LISyC : www.lisyc.univ-brest.fr

Le laboratoire de recherche en informatique de l'ENIB est le LISyC (Laboratoire d'Informatique des Systèmes Complexes). Le LISyC est à Brest une Equipe d'Accueil (EA 3883) commune à l'Université de Bretagne Occidentale (UBO), à l'Ecole Nationale d'Ingénieurs de Brest (ENIB) et à l'Ecole Nationale Supérieure des Ingénieurs des Etudes des Techniques d'Armement (ENSIETA).

Le LISyC a fait sienne l'idée selon laquelle *comprendre et maîtriser les comportements des systèmes complexes, naturels ou artificiels, constituent pour les scientifiques le défi majeur du 21ème siècle.* Il compte actuellement 50 enseignants-chercheurs permanents et 25 doctorants

regroupés en 4 équipes STIC et 1 équipe SHS qui explorent de manière complémentaire cette problématique :

ARéVi	: Ateliers de Réalité Virtuelle
in virtuo	: in virtuo
IdM	: Ingénierie des Modèles
SARA	: Simulation, Apprentissages, Représentations, Action
SuSy	: Sûreté des Systèmes

Les équipes ARéVi, in virtuo et SARA sont localisées au CERV : le Centre Européen de Réalité Virtuelle, établissement de l'ENIB ouvert en juin 2004.

CERV : www.cerv.fr



Chapitre 2

Instructions de base



Sommaire

2.1 Introduction

Un algorithme est une suite ordonnée d'instructions qui indique la démarche à suivre pour résoudre une série de problèmes équivalents. Ainsi quand on définit un algorithme, celui-ci ne doit contenir que des instructions compréhensibles par celui qui devra l'exécuter. Dans ce cours, nous devrons donc apprendre à définir des algorithmes pour qu'ils soient compréhensibles — et donc exécutables — par un ordinateur.

2.1.1 Jeu d'instructions

Chaque microprocesseur a son jeu d'instructions de base dont le nombre varie typiquement de quelques dizaines à quelques centaines selon le type d'architecture du processeur. On peut classer ces instructions de base en 5 grands groupes : les opérations arithmétiques (`+`, `-`, `*`, `/`...), les opérations logiques (`not`, `and`, `or`...), les instructions de transferts de données (`load`, `store`, `move`...), les instructions de contrôle du flux d'instructions (branchements impératifs et conditionnels, boucles, appels de procédure...), et les instructions d'entrée-sortie (`read`, `write`...). Le traitement des ces instructions par le microprocesseur passe ensuite par 5 étapes :

1. `fetch` : chargement depuis la mémoire de la prochaine instruction à exécuter,
2. `decode` : décodage de l'instruction,
3. `load operand` : chargement des données nécessaires à l'instruction,
4. `execute` : exécution de l'instruction,
5. `result write back` : mise à jour du résultat dans un registre ou en mémoire.

Le langage machine est le langage compris par le microprocesseur. Ce langage est difficile à maîtriser puisque chaque instruction est codée par une séquence donnée de bits. Afin de faciliter la tâche du programmeur, on a d'abord créé le langage assembleur qui utilise des mnémoniques pour le codage des instructions puis les langages de plus haut niveau d'expressivité (FORTRAN, C, JAVA, PYTHON...). Le tableau ci-dessous compare les codes équivalents pour décrire l'addition de 2 entiers dans différents langages informatiques : le langage machine, le langage assembleur, le langage PASCAL et le langage PYTHON. On constate sur cet exemple une évolution progressive

Remarque 2.1 : *On distingue classiquement 2 grands types d'architectures de micro-processeurs :*
- *les architectures* RISC (Reduced Instruction Set Computer) *préconisent un petit nombre d'instructions élémentaires dans un format fixe ;*
- *les architectures* CISC (Complex Instruction Set Computer) *sont basées sur des jeux d'instructions très riches de taille variable offrant des instructions composées de plus haut niveau d'abstraction.*

Chaque architecture possède ses avantages et ses inconvénients : pour le RISC *la complexité est reportée au niveau du compilateur, pour le* CISC *le décodage est plus pénalisant. En fait les machines* CISC *se sont orientées vers une architecture* RISC *où les instructions* CISC *sont traduites en instructions* RISC *traitées par le coeur du processeur.*

Remarque 2.2 : *Le coeur du microprocesseur est régulé par un quartz qui oscille avec une fréquence exprimée en Hz. Le temps de cycle est l'inverse de la fréquence. Ainsi pour une fréquence de 100 MHz, on a un temps de cycle de 10 ns. L'exécution d'une instruction nécessite plusieurs temps de cycle, c'est ce que l'on appelle le* CPI (Cycles per Instruction).

du pouvoir d'expressivité des langages, du langage machine aux langages de haut niveau.

machine	assembleur	PASCAL	PYTHON
A1 00 01 8B 1E 02 01 01 D8 A3 04 01	MOV AX,[100h] MOV BX,[102h] ADD AX,BX MOV [104h],AX	var a,b,c : integer ; c := a + b ;	c = a + b

2.1.2 Instructions de base

Dans ce cours, nous nous intéresserons aux instructions disponibles dans un langage de haut niveau tel que PYTHON. Les principales instructions (*statements*) concerneront l'affectation (*assignment*), les tests (*conditional statements*) et les boucles (*loops*). Le tableau ci-dessous donne la syntaxe PYTHON des instructions de base qui seront utilisées dans ce chapitre (d'après [10]). Une liste plus détaillée des principales instructions en PYTHON est proposée en annexe 2.6.2 page 92.



PYTHON : www.python.org

Remarque 2.3 : *L'anglais est la langue couramment utilisée en informatique. Il est absolument essentiel de lire l'anglais technique sans problème. Vous devez être capable de traduire le tableau ci-contre extrait sans traduction d'une référence en anglais [10].*

Statement	Result
`pass`	Null statement
`print([s1] [, s2 ]*)`	Writes to `sys.stdout`. Puts spaces between arguments `si`. Puts newline at end unless arguments end with `end=` (ie : `end=' '`). `print` is not required when running interactively, simply typing an expression will print its value, unless the value is `None`.
`a = b`	Basic assignment - assign object `b` to label `a`

`if condition :`
`  suite`
`[elif condition : suite]*`
`[else :`
`  suite]`
	Usual `if/else if/else` statement.

`while condition :` `  suite`	Usual `while` statement.
`for element in sequence :` `  suite`	Iterates over `sequence`, assigning each element to `element`. Use built-in `range` function to iterate a number of times.

2.2 Affectation

2.2.1 Variables

Exemple 2.1 : La température Fahrenheit
Le degré Fahrenheit (°F) est une unité de mesure de la température, qui doit son nom au physicien allemand Daniel Gabriel Fahrenheit (1686-1736), qui la proposa en 1724. Dans l'échelle de température de Fahrenheit, le point de solidification de l'eau est de 32 degrés, et son point d'ébullition de 212 degrés. Ainsi par exemple, $70°F$ *correspondent approximativement à* $21°C$.

Pour effectuer la conversion Fahrenheit → Celcius, nous commençons par donner un nom à la température Fahrenheit, par exemple t_F, ainsi qu'à la température Celcius, par exemple t_C. Puis nous pouvons exprimer la relation générale qui lie t_C à t_F : $t_C = \frac{5}{9}(t_F - 32)$ et appliquer cette relation dans un cas particulier, par exemple $t_F = 70$: $t_C = \frac{5}{9}(70 - 32) \approx 21$. ■TD2.1

En informatique, l'essentiel du travail effectué par un programme d'ordinateur consiste à manipuler des données. Ces données peuvent être très diverses (par exemple des températures) et pour accéder à ces données, il est pratique de les nommer plutôt que de connaître explicitement leur adresse en mémoire.

Une donnée apparaît ainsi sous un nom de variable (par exemple `tC` ou `tF`) : on dit que la variable dénote une valeur (figure 2.1). Pour la machine, il s'agit d'une référence désignant une adresse mémoire, c'est-à-dire un emplacement précis dans la mémoire vive où est stockée une valeur bien déterminée qui est la donnée proprement dite.

Définition 2.1 : variable
Une variable est un objet informatique qui associe un nom à une valeur qui peut éventuellement varier au cours du temps.

Les noms de variables sont des identificateurs arbitraires, de préférence assez courts mais aussi explicites que possible, de manière à exprimer clairement ce que la variable est censée référencer (la sémantique de la donnée référencée par la variable). Les noms des variables doivent en outre obéir à quelques règles simples :

– Un nom de variable est une séquence de lettres (a...z , A...Z) et de chiffres (0...9), qui doit toujours commencer par une lettre.

TD 2.1 : Unité de pression
Le torr (torr) ou millimètre de mercure (mmHg) est une unité de mesure de la pression qui tire son nom du physicien et mathématicien italien Evangelista Torricelli (1608-1647). Il est défini comme la pression exercée à 0°C par une colonne de 1 millimètre de mercure (mmHg). Il a plus tard été indexée sur la pression atmosphérique : 1 atmosphère normale correspond à 760 mmHg et a 101 325 Pa.
Ecrire une instruction qui permette de passer directement des torrs au pascals (Pa).

Fig. 2.1 : Définition de l'académie (4)
DÉNOTER *v. tr. XIVe siècle. Emprunté du latin denotare, « désigner, faire connaître ». 1. Indiquer comme caractéristique, signifier, révéler. 2. LOGIQUE. Désigner la totalité des objets dont les caractères sont fixés par un concept.*

Remarque 2.4 : *Une variable peut être vue comme une case en mémoire vive, que le programme va repérer par une étiquette (une adresse ou un nom). Pour avoir accès au contenu de la case (la valeur de la variable), il suffit de la désigner par son étiquette : c'est-à-dire soit par son adresse en mémoire, soit par son nom.*

Remarque 2.5 : *En mathématiques, une « variable » est généralement une inconnue, qui recouvre un nombre non précisé de valeurs. En informatique, une variable possède à un moment donné une valeur et une seule.*

– Seules les lettres ordinaires sont autorisées. Les lettres accentuées, les cédilles, les espaces, les caractères spéciaux tels que `$`, `#`, `@`, etc. sont interdits, à l'exception du caractère _ (souligné).
– La « casse » est significative : les caractères majuscules et minuscules sont distingués. Ainsi, `python`, `Python`, `PYTHON` sont des variables différentes.
– Par convention, on écrira l'essentiel des noms de variable en caractères minuscules (y compris la première lettre). On n'utilisera les majuscules qu'à l'intérieur même du nom pour en augmenter éventuellement la lisibilité, comme dans `programmePython` ou `angleRotation`. Une variable dont la valeur associée ne varie pas au cours du programme (on parle alors de constante) pourra être écrite entièrement en majuscule, par exemple `PI` ($\pi = 3.14$).
– Le langage lui-même peut se réserver quelques noms comme c'est le cas pour PYTHON (figure 3.5). Ces mots réservés ne peuvent donc pas être utilisés comme noms de variable.

Fig. 2.2 : MOTS RÉSERVÉS EN PYTHON

```
and        del       for      is       raise
assert     elif      from     lambda   return
break      else      global   not      try
class      except    if       or       while
continue   exec      import   pass     with
def        finally   in       print    yield
```

2.2.2 Attribuer une valeur

Une fois nommée, il est souvent nécessaire de modifier la valeur de la donnée référencée par une variable. C'est le rôle de l'instruction d'affectation.

Définition 2.2 : AFFECTATION
L'affectation est l'opération qui consiste à attribuer une valeur à une variable.

L'instruction d'affectation est notée `=` en PYTHON : `variable = valeur`. Le nom de la variable à modifier est placé dans le membre de gauche du signe `=`, la valeur qu'on veut lui attribuer dans le membre de droite. Le membre de droite de l'affectation est d'abord évalué sans être modifié puis la valeur obtenue est affectée à la variable dont le nom est donné dans le membre de gauche de l'affectation ; ainsi, cette opération ne modifie que le membre de gauche de l'affectation. Le membre de droite peut être une constante ou une expression évaluable.

`variable = constante` : La constante peut être d'un type quelconque (figure 2.3) : entier, réel, booléen, chaîne de caractères, tableau, matrice, dictionnaire... comme le suggèrent les exemples suivants :

Fig. 2.3 : TYPES DE BASE EN PYTHON

type	*nom*	*exemples*
booléens	`bool`	`False, True`
entiers	`int`	`3, -7`
réels	`float`	`3.14, 7.43e-3`
chaînes	`str`	`'salut', "l'eau"`
n-uplets	`tuple`	`1,2,3`
listes	`list`	`[1,2,3]`
dictionnaires	`dict`	`{'a' :4, 'r' :8}`

TD 2.2 : Suite arithmétique (1)
Ecrire une instruction qui calcule la somme $s = \sum_0^n u_k$ des n premiers termes d'une suite arithmétique $u_k = a + r \cdot k$.

Fig. 2.4 : Définition de l'académie (5)
INCRÉMENT *n. m. XVe siècle, encrement. Emprunté du latin incrementum, « accroissement ». INFORM. Quantité fixe dont on augmente la valeur d'une variable à chaque phase de l'exécution du programme.*
DÉCRÉMENT *n. m. XIXe siècle. Emprunté de l'anglais decrement, du latin decrementum, « amoindrissement, diminPrincipales affectations en* Python*ution ». MATH. INFORM. Quantité fixe dont une grandeur diminue à chaque cycle.*

Remarque 2.6 : *Avec l'exemple de l'incrémentation (*`i = i + 1`*), on constate que l'affectation est une opération typiquement informatique qui se distingue de l'égalité mathématique. En effet, en mathématique une expression du type* `i = i+1` *se réduit en* `0 = 1` *! Alors qu'en informatique, l'expression* `i = i+1` *conduit à ajouter* `1` *à la valeur de* `i` *(évaluation de l'expression* `i+1`*), puis à donner cette nouvelle valeur à* `i` *(affectation).*

Fig. 2.5 : Principales affectations en Python

a = b		
a += b	≡	a = a + b
a -= b	≡	a = a - b
a *= b	≡	a = a * b
a /= b	≡	a = a / b
a %= b	≡	a = a % b
a **= b	≡	a = a ** b

```
booleen = False              autreBooleen = True
entier = 3                   autreEntier = -329
reel = 0.0                   autreReel = -5.4687e-2
chaine = "salut"             autreChaine = 'bonjour, comment ça va ?'
tableau = [5,2,9,3]          autreTableau = ['a',[6,3.14],[x,y,[z,t]]]
matrice = [[1,2],[6,7]]      autreMatrice = [[1,2],[3,4],[5,6],[7,8]]
nUplet = 4,5,6               autreNUplet = "e",True,6.7,3,"z"
dictionnaire = {}            autreDictionnaire = {"a":7, "r":-8}
```

`variable = expression` : L'expression peut être n'importe quelle expression évaluable telle qu'une opération logique (`x = True or False and not True`), une opération arithmétique (`x = 3 + 2*9 - 6*7`), un appel de fonction (`y = sin(x)`) ou toute autre combinaison évaluable (`x = (x != y) and (z + t >= y) or (sin(x) < 0)`). ■TD2.2

```
reste = a%b               quotient = a/b
somme = n*(n+1)/2         sommeGeometrique =  s = a*(b**(n+1) - 1)/(b-1)
delta = b*b - 4*a*c       racine = (-b + sqrt(delta))/(2*a)
surface = pi*r**2         volume = surface * hauteur
```

L'expression du membre de droite peut faire intervenir la variable du membre de gauche comme dans `i = i + 1`. Dans cet exemple, on évalue d'abord le membre de droite (`i + 1`) puis on attribue la valeur obtenue au membre de gauche (`i`) ; ainsi, à la fin de cette affectation, la valeur de `i` a été augmentée de `1` : on dit que `i` a été incrémenté de `1` (figure 2.4) et on parle d'incrémentation de la variable `i` (remarque 2.6). Python propose un opérateur d'incrémentation (`+=`) et d'autres opérateurs d'affectation qui peuvent toujours se ramener à l'utilisation de l'opérateur `=`, l'opérateur d'affectation de base (figure 2.5).

L'affectation a ainsi pour effet de réaliser plusieurs opérations dans la mémoire de l'ordinateur :

- créer et mémoriser un nom de variable,
- lui attribuer un type bien déterminé,
- créer et mémoriser une valeur particulière,
- établir un lien (par un système interne de pointeurs) entre le nom de la variable et l'emplacement mémoire de la valeur correspondante.

2.2.3 Séquences d'affectations

Exemple 2.2 : Permutation de 2 nombres
Un apprenti informaticien a qui on demandait d'échanger (swap) les valeurs de 2 variables `x` *et* `y` *proposa la suite d'instructions suivante :*

```
x = y
y = x
```

et eut la désagréable surprise de constater que les valeurs des variables n'étaient pas permutées après cette séquence d'affectations.

En effet, pour fixer les idées supposons qu'initialement `x = 10` et `y = 20`. L'affectation `x = y` conduit à évaluer `y` puis à attribuer la valeur de `y` (`20`) à `x` : `x` vaut maintenant `20`. La deuxième affectation (`y = x`) commence par évaluer `x` puis à attribuer la valeur de `x` (`20` !) à `y`. Après ces 2 affectations, `x` et `y` sont donc identiques et non permutées ! Pour effectuer la permutation, l'apprenti informaticien aurait pu utiliser une variable temporaire (que nous nommerons `tmp`) et exécuter la séquence d'instructions suivante :

```
tmp = x
x = y
y = tmp
```

La première affectation (`tmp = x`) permet de stocker la valeur initiale de `x` (`10`), la deuxième (`x = y`) attribue à `x` la valeur de `y` (`20`) et la troisième (`y = tmp`) attribue à `y` la valeur de `tmp`, c'est-à-dire la valeur initiale de `x` (`10`). Ainsi, les valeurs finales de `x` et `y` (`20` et `10`) sont bien permutées par rapport aux valeurs initiales (`10` et `20`).

■ TD2.3

Remarque 2.7 : *L'affectation n'est pas une opération commutative (symétrique) :* `a = b` $\neq$ `b = a`. *En effet, avec l'instruction* `a = b` *on modifie la valeur de* `a` *et pas celle de* `b` *tandis qu'avec l'instruction* `b = a`*, on modifie* `b` *mais pas* `a`*.*

Remarque 2.8 : *En* Python*, les n-uplets permettent d'écrire plus simplement la permutation de variables :*

```
x, y = y, x
```

TD 2.3 : Permutation circulaire (1)
Effectuer une permutation circulaire droite entre les valeurs de 4 entiers x, y, z et t.

Exemple 2.3 : Un calcul de pgcd (1)
Le plus grand commun diviseur de 2 entiers a et b peut se calculer en appliquant la relation de récurrence $\text{pgcd}(a, b) = \text{pgcd}(b, a\%b)$ *si $b \neq 0$ jusqu'à ce que le reste ($a\%b$) soit nul* ($\text{pgcd}(d, 0) = d$ *si $d \neq 0$).*

Ainsi, pour calculer le pgcd de $a = 12$ et de $b = 18$, on applique 3 fois de suite cette relation : $\text{pgcd}(a, b) = \text{pgcd}(b, a\%b) \Rightarrow \text{pgcd}(12, 18) = \text{pgcd}(18, 12) = \text{pgcd}(12, 6) = \text{pgcd}(6, 0) = 6$. Ce qui peut se traduire en Python par la séquence d'affectations suivante :

```
a = 12      # a = 12        a = b     # a = 12
b = 18      # b = 18        b = r     # b = 6
r = a%b     # r = 12        r = a%b   # r = 0
a = b       # a = 18        a = b     # a = 6
b = r       # b = 12        b = r     # b = 0
r = a%b     # r = 6
```

A la fin de la séquence, on a $a = 6$ et $b = 0$: a est le pgcd recherché.

■ TD2.4

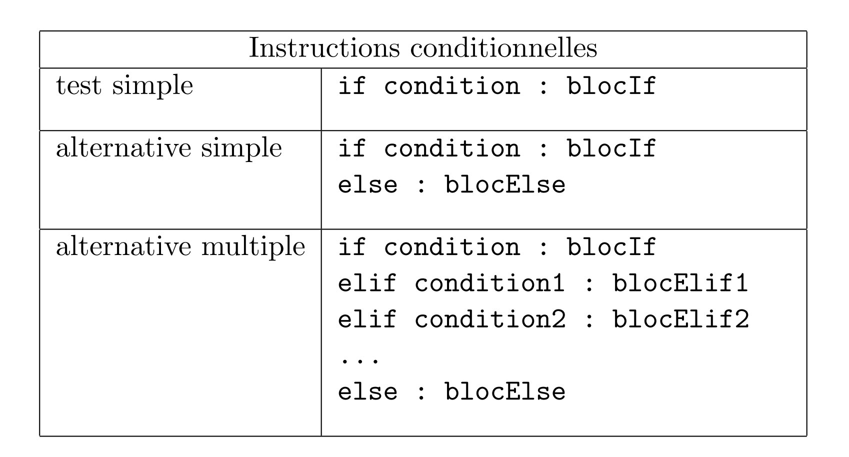
TD 2.4 : Séquence d'affectations (1)
Quelles sont les valeurs des variables a, b, q et r après la séquence d'affectations suivante ?

```
a = 19
b = 6
q = 0
r = a
r = r - b
q = q + 1
r = r - b
q = q + 1
r = r - b
q = q + 1
```

Les 2 exemples 2.2 et 2.3 précédents illustrent la possibilité de réaliser des calculs plus ou moins compliqués à l'aide d'une séquence d'affectations bien choisies. Mais ce sont les tests et les boucles qui nous permettront d'aborder des algorithmes réutilisables, et plus robustes, en améliorant l'expressivité du programmeur.

2.3 Tests

Sauf mention explicite, les instructions d'un algorithme s'exécutent les unes après les autres, dans l'ordre où elles ont été écrites. Le « chemin » suivi à travers un algorithme est appelé le flux d'instructions (figure 2.6), et les constructions qui le modifient sont appelées des instructions de contrôle de flux. On exécute normalement les instructions de la première à la dernière, sauf lorsqu'on rencontre une instruction de contrôle de flux : de telles instructions vont permettre de suivre différents chemins suivant les circonstances. C'est en particulier le cas de l'instruction conditionnelle qui n'exécute une instruction que sous certaines conditions préalables. Nous distinguerons ici 3 variantes d'instructions conditionnelles (figure 2.7) :



Instructions conditionnelles	
test simple	`if condition : blocIf`
alternative simple	`if condition : blocIf` `else : blocElse`

alternative multiple	`if condition : blocIf`
	`elif condition1 : blocElif1`
	`elif condition2 : blocElif2`
	`...`
	`else : blocElse`

où `if`, `else` et `elif` sont des mots réservés, `condition` une expression booléenne (à valeur `True` ou `False`) et `bloc...` un bloc d'instructions.

Fig. 2.6 : Flux d'instructions

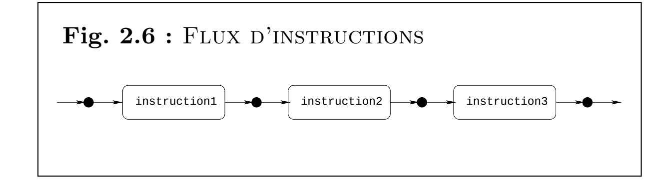


Fig. 2.7 : Définition de l'académie (6)
ALTERNATIVE *n. f. XVe siècle, comme terme de droit ecclésiastique ; XVIIe siècle, au sens moderne. Forme féminine substantivée d'alternatif. Choix nécessaire entre deux propositions, deux attitudes dont l'une exclut l'autre.*

Remarque 2.9 : *A propos des instructions conditionnelles, on parle souvent des instructions « `if` » dans le jargon des informaticiens.*

Remarque 2.10 : `elif ...` *et la contraction de* `else : if ...`.

2.3.1 Tests simples

L'instruction « `if` » sous sa forme la plus simple (figure 2.8) permet de tester la validité d'une condition. Si la condition est vraie, alors le bloc d'instructions `blocIf` après le « `:` » est exécuté. Si la condition est fausse, on passe à l'instruction suivante dans le flux d'instructions.

Définition 2.3 : TEST SIMPLE
Le test simple est une instruction de contrôle du flux d'instructions qui permet d'exécuter une instruction sous condition préalable.

La condition évaluée après l'instruction « `if` » est donc une expression booléenne qui prend soit la valeur `False` (faux) soit la valeur `True` (vrai). Elle peut contenir les opérateurs de comparaison suivants :

```
x == y    # x est égal à y
x != y    # x est différent de y
x > y     # x est plus grand que y
x < y     # x est plus petit que y
x >= y    # x est plus grand que, ou égal à y
x <= y    # x est plus petit que, ou égal à y
```

Mais certains problèmes exigent parfois de formuler des conditions qui ne peuvent pas être exprimées sous la forme d'une simple comparaison. Par exemple, la condition $x \in [0, 1[$ s'exprime par la combinaison de deux conditions $x \geq 0$ et $x < 1$ qui doivent être vérifiées en même temps. Pour combiner ces conditions, on utilise les opérateurs logiques `not`, `and` et `or` (figure 2.9). Ainsi la condition $x \in [0, 1[$ pourra s'écrire en PYTHON : `(x >= 0) and (x < 1)`. ■TD2.5

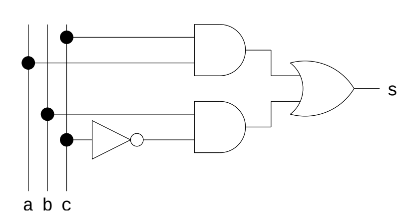
Le tableau ci-dessous donne les tables de vérité des opérateurs `not`, `or` et `and`, leur représentation graphique traditionnelle ainsi que leurs principales propriétés. ■TD2.6

Fig. 2.8 : LE TEST SIMPLE
`if condition : blocIf`

Fig. 2.9 : PRINCIPAUX OPÉRATEURS PYTHON
Opérateurs logiques : `not a`, `a and b`, `a or b`
Opérateurs de comparaison : `x == y`, `x != y`, `x < y`, `x <= y`, `x > y`, `x >= y`
Opérateurs arithmétiques : `+x`, `-x`, `x + y`, `x - y`, `x * y`, `x / y`, `x % y`, `x**y`

TD 2.5 : OPÉRATEURS BOOLÉENS DÉRIVÉS (1)
En utilisant les opérateurs booléens de base (`not`*,* `and` *et* `or`*), ecrire un algorithme qui affecte successivement à une variable* `s` *le résultat des opérations booléennes suivantes : ou exclusif* (xor, $a \oplus b$)*, non ou* (nor, $\overline{a + b}$)*, non et* (nand, $\overline{a \cdot b}$)*, implication* ($a \Rightarrow b$) *et équivalence* ($a \Leftrightarrow b$)*.*

TD 2.6 : CIRCUIT LOGIQUE (1)
Donner les séquences d'affectations permettant de calculer la sortie s du circuit logique suivant en fonction de ses entrées a, b et c.



négation	disjonction	conjonction
`not a`	`a or b`	`a and b`
	`not (a or b)`	`not (a and b)`

a	$\overline{a}$
0	1
1	0

a	b	$a+b$
0	0	0
0	1	1
1	0	1
1	1	1

a	b	$a \cdot b$
0	0	0
0	1	0
1	0	0
1	1	1

$\forall a, b, c \in \{0; 1\}$:

$a + 0 = a \qquad a \cdot 1 = a$

$a + 1 = 1 \qquad a \cdot 0 = 0$

$a + a = a \qquad a \cdot a = a$

$a + \overline{a} = 1 \qquad a \cdot \overline{a} = 0$

$a + (a \cdot b) = a \qquad a \cdot (a + b) = a$

$\overline{\overline{a}} = a \qquad \overline{a + b} = \overline{a} \cdot \overline{b} \qquad \overline{a \cdot b} = \overline{a} + \overline{b}$

$(a + b) = (b + a) \qquad (a \cdot b) = (b \cdot a)$

$(a + b) + c = a + (b + c)$

$(a \cdot b) \cdot c = a \cdot (b \cdot c)$

$a + (b \cdot c) = (a + b) \cdot (a + c)$

$a \cdot (b + c) = (a \cdot b) + (a \cdot c)$

■TD2.7

TD 2.7 : Lois de De Morgan
Démontrer à l'aide des tables de vérité les lois de De Morgan $\forall a, b \in \{0; 1\}$ *:*

1. $\overline{(a + b)} = \overline{a} \cdot \overline{b}$
2. $\overline{(a \cdot b)} = \overline{a} + \overline{b}$

2.3.2 Alternatives simples

Exemple 2.4 : Extrait d'un dialogue entre un conducteur égaré et un piéton

– *Pourriez-vous m'indiquer le chemin de la gare, s'il vous plait ?*
– *Oui bien sûr : vous allez tout droit jusqu'au prochain carrefour. Si la rue à droite est autorisée à la circulation — hier elle était en travaux — alors prenez la et ensuite c'est la deuxième à gauche et vous verrez la gare. Sinon, au carrefour, vous allez tout droit et vous prenez la première à droite, puis encore la première à droite et vous y êtes.*
– *Merci.*

L'algorithme décrit par le piéton propose une alternative entre deux solutions. Le conducteur égaré devra tester si la rue est en travaux avant de prendre la décision d'aller à droite au carrefour ou de continuer tout droit. En algorithmique, un tel choix est proposé par l'alternative simple, instruction conditionnelle dite « `if ... else` ».

L'instruction « `if ... else` » teste une condition (figure 2.10). Si la condition est vraie, alors le bloc d'instructions `blocIf` après le « `:` » est exécuté. Si la condition est fausse, c'est le bloc d'instructions `blocElse` après le « `else :` » (sinon) qui est exécuté. Seul l'un des 2 blocs est donc exécuté.

Fig. 2.10 : L'alternative simple

```
if condition : blocIf
else : blocElse
```

Remarque 2.11 : *Le test simple (figure 2.8 page 47) est équivalent à une alternative simple où on explicite le fait de ne rien faire (instruction* `pass`*) dans le bloc d'instructions associé au* `else` *:*

```
if condition : bloc
else : pass
```

voir également le TD 2.30 page 70.

Définition 2.4 : ALTERNATIVE SIMPLE

L'alternative simple est une instruction de contrôle du flux d'instructions qui permet de choisir entre deux instructions selon qu'une condition est vérifiée ou non.

Exemple 2.5 : VALEUR ABSOLUE D'UN NOMBRE

L'algorithme qui détermine la valeur absolue y d'un nombre x peut s'écrire de la manière suivante :

```
if x < 0 : y = -x
else : y = x
```

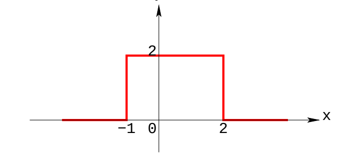
On commence par tester le signe de x (`if x < 0`), si $x < 0$, alors la valeur absolue y vaut $-x$ (`: y = -x`) sinon ($x \geq 0$) elle vaut x (`else : y = x`). ■TD2.8

Exemple 2.6 : FONCTION « PORTE »

On considère la fonction « porte » f dont le graphe est donné ci-contre. L'alternative suivante permet de calculer $y = f(x)$:

```
if x < -1 or x > 2 : y = 0
else : y = 2
```

■TD2.9



Les exemples 2.4 et 2.6 précédents nous montrent qu'une alternative se comporte comme un aiguillage de chemin de fer dans le flux d'instructions (figure 2.11). Un « `if ... else` » ouvre deux voies correspondant à deux traitements différents, et seule une de ces voies sera empruntée (un seul des deux traitements est exécuté). Mais il y a des situations où deux voies ne suffisent pas : on utilise alors des alternatives simples en cascade (ou alternatives multiples).

2.3.3 Alternatives multiples

Exemple 2.7 : ETAT DE L'EAU

A pression ambiante, l'eau est sous forme de glace si la température est inférieure à $0°C$, sous forme de liquide si la température est comprise entre $0°C$ et $100°C$ et sous forme de vapeur au-delà de $100°C$.

Un algorithme qui devrait déterminer l'état de l'eau en fonction de la température doit pouvoir choisir entre trois réponses possibles : solide, liquide ou vapeur. Une première version de cet

TD 2.8 : MAXIMUM DE 2 NOMBRES

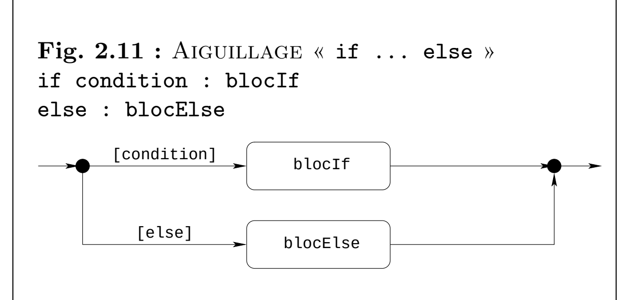
Ecrire un algorithme qui détermine le maximum m de 2 nombres x et y.

TD 2.9 : FONCTION « PORTE »

Proposer une autre alternative simple pour calculer la fonction « porte » de l'exemple 2.6 ci-contre.

Fig. 2.11 : AIGUILLAGE « `if ... else` »

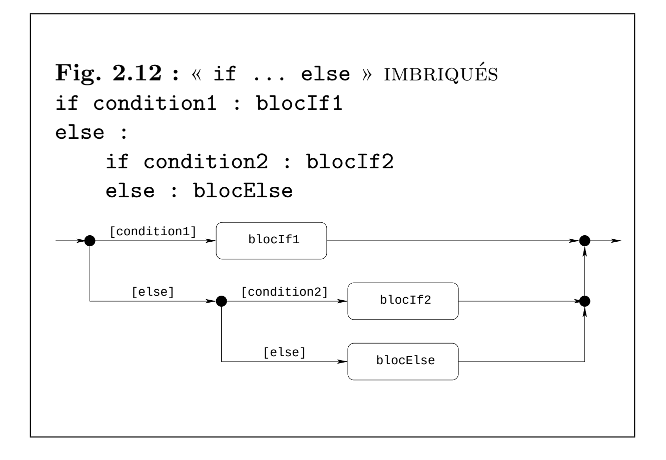
```
if condition : blocIf
else : blocElse
```



L'étiquette `[condition]` *signifie qu'on passe par la voie correspondante si la condition est vérifiée (*`True`*), sinon on passe par la voie étiquetée* `[else]`.

Fig. 2.12 : « `if ... else` » IMBRIQUÉS

```
if condition1 : blocIf1
else :
    if condition2 : blocIf2
    else : blocElse
```



TD 2.10 : Ouverture d'un guichet
A l'aide d'alternatives simples imbriquées, écrire un algorithme qui détermine si un guichet est `'ouvert'` *ou* `'fermé'` *selon les jours de la semaine (*`'lundi'`*,* `'mardi'`*, ... ,*`'dimanche'`*) et l'heure de la journée (entre 0h et 24h). Le guichet est ouvert tous les jours de 8h à 13h et de 14h à 17h sauf le samedi après-midi et toute la journée du dimanche.*

Fig. 2.13 : L'alternative multiple

```
if condition : blocIf
elif condition1 : blocElif1
elif condition2 : blocElif2
...
else : blocElse
```


L'alternative multiple ci-dessus est équivalente à un ensemble d'alternatives simples imbriquées :

```
if condition : blocIf
else :
    if condition1 : blocElif1
    else :
        if condition2 : blocElif2
        else :
         ..
            else : blocElse
```

algorithme utilise 3 tests simples :

```
 if t < 0 : eau = 'glace'
if t >= 0 and t <= 100 : eau = 'liquide'
if t > 100 : eau = 'vapeur'
```

Cet algorithme est correct mais va évaluer les 3 conditions qui portent sur la même variable `t` et qui sont exclusives les unes des autres ; en effet, si `(t < 0)`, alors on ne peut pas avoir `(t >= 0 and t <= 100)` ni `(t > 100)`. Il est donc inutile d'évaluer les 2 dernières conditions si la première est vérifiée, ou d'évaluer la dernière condition si la deuxième est vérifiée. On préfère donc imbriquer les tests de la manière suivante :

```
if t < 0 : eau = 'glace'
else :
    if t <= 100 : eau = 'liquide'
    else : eau = 'vapeur'
```

■TD2.10

On commence par évaluer la première condition `(t < 0)`. Si la condition est vérifiée, on exécute l'affectation `eau = 'glace'` ; sinon `(t >= 0)`, on évalue la deuxième condition `(t <= 100)` qui en fait est équivalente à `(t >= 0) and (t <= 100)`. Si la condition est vérifiée, on exécute l'affectation `eau = 'liquide'` ; sinon `(t > 100)`, on exécute l'affectation `eau = 'vapeur'`. La figure 2.12 illustre le contrôle du flux d'instructions lors de deux « `if ... else` » imbriqués : il s'agit de deux aiguillages en cascade. Afin de simplifier l'écriture des tests imbriqués, on peut contracter le « `else : if` » en `elif` et obtenir une version plus compacte de l'algorithme, strictement équivalente à la version précédente :

```
if t < 0 : eau = 'glace'
elif t <= 100 : eau = 'liquide'
else : eau = 'vapeur'
```

L'instruction « `if ... elif` » teste une première condition (figure 2.13). Si cette condition est vraie, alors le bloc d'instructions `blocIf` est exécuté. Si la première condition est fausse, on teste la deuxième (`condition1`). Si la deuxième condition est vérifiée, c'est le bloc d'instructions `blocElif1` après le premier « `elif :` » (sinon-si) qui est exécuté ; sinon on teste la condition suivante (`condition2`). Si elle est vérifiée, c'est le bloc d'instructions `blocElif2` après le deuxième « `elif :` » qui est exécuté et ainsi de suite. Si aucune des conditions n'est vérifiée, c'est le bloc d'instructions `blocElse` qui est exécuté. Dans tous les cas, un seul des blocs est donc exécuté.

Définition 2.5 : ALTERNATIVE MULTIPLE
L'alternative multiple est une instruction de contrôle du flux d'instructions qui permet de choisir entre plusieurs instructions en cascadant des alternatives simples.

Exemple 2.8 : MENTIONS DU BACCALAURÉAT
Au baccalauréat, la mention associée à une note sur 20 est `'très bien'` *pour les notes supérieures ou égales à 16,* `'bien'` *pour les notes comprises entre 14 inclus et 16 exclu,* `'assez bien'` *pour les notes comprises entre 12 inclus et 14 exclu,* `'passable'` *pour les notes comprises entre 10 inclus et 12 exclu et* `'insuffisant'` *pour les notes strictement inférieures à 10.*

On peut utiliser une alternative multiple pour déterminer la mention au bac en fonction de la note :

```
if note < 10 : mention = 'insuffisant'
elif note < 12 : mention = 'passable'
elif note < 14 : mention = 'assez bien'
elif note < 16 : mention = 'bien'
else : mention = 'très bien'
```

■ TD2.11

TD 2.11 : CATÉGORIE SPORTIVE
Ecrire un algorithme qui détermine la catégorie sportive d'un enfant selon son âge :
- *Poussin de 6 à 7 ans,*
- *Pupille de 8 à 9 ans,*
- *Minime de 10 à 11 ans,*
- *Cadet de 12 ans à 14 ans.*

2.4 Boucles

Exemple 2.9 : UN CALCUL DE PGCD (2)
Le plus grand commun diviseur de 2 entiers a et b peut se calculer en appliquant la relation de récurrence $\text{pgcd}(a, b) = \text{pgcd}(b, a\%b)$ *jusqu'à ce que le reste ($a\%b$) soit nul.*
Dans l'exemple 2.3 page 45, pour calculer le pgcd de $a = 12$ et de $b = 18$, on appliquait 3 fois de suite cette relation : $\text{pgcd}(12, 18) = \text{pgcd}(18, 12) = \text{pgcd}(12, 6) = \text{pgcd}(6, 0) = 6$. *L'algorithme correspondant faisait donc apparaître 3 fois de suite les mêmes instructions après l'initialisation des variables a et b :*

```
r = a%b
a = b
b = r
```

Si nous voulons maintenant calculer le pgcd de 44 *et* 5648, *il nous faudra répéter 5 fois la même séquence d'instructions pour trouver que* $\text{pgcd}(44, 5648) = 4$.

Ce nouvel exemple de calcul de pgcd soulève au moins 2 questions :

1. Comment éviter de répéter explicitement plusieurs fois de suite la même séquence d'instructions ?
2. Comment éviter de savoir à l'avance combien de fois il faut répéter la séquence pour obtenir le bon résultat ?

De nouvelles instructions de contrôle de flux sont introduites pour répondre à ces questions : les instructions itératives. On parle également de boucles, de répétitions ou encore d'itérations (figure 2.14). Nous distinguerons 2 variantes d'instructions itératives :

Instructions itératives	
itération conditionnelle	`while condition : blocWhile`
parcours de séquence	`for element in sequence : blocFor`

où `while`, `for` et `in` sont des mots réservés, `condition` une expression booléenne (à valeur `True` ou `False`), `element` un élément de la séquence `sequence` et `bloc...` un bloc d'instructions.

Fig. 2.14 : Définition de l'académie (7)
ITÉRATION *n. f. XVe siècle. Emprunté du latin iteratio, « répétition, redite ». Répétition. MATH. Répétition d'un calcul avec modification de la variable, qui permet d'obtenir par approximations successives un résultat satisfaisant.*

Remarque 2.12 : *A propos des instructions itératives, on parle souvent des boucles « `while` » ou des boucles « `for` » dans le jargon des informaticiens.*

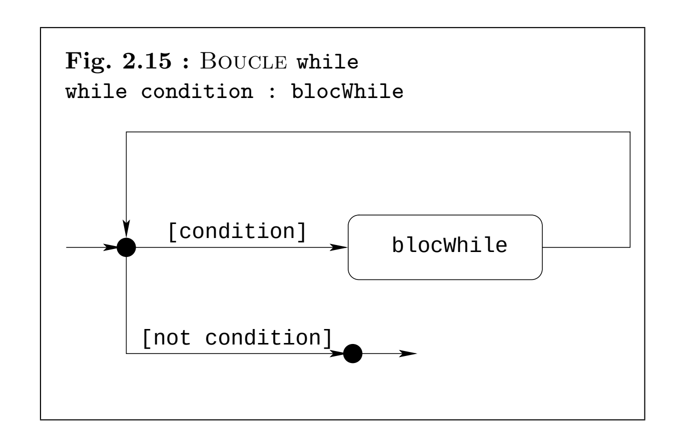
2.4.1 Itération conditionnelle

L'instruction « `while` » permet de répéter plusieurs fois une même instruction (figure 2.15) : le bloc d'instructions `blocWhile` est exécuté tant que (*while*) la condition est vérifiée. On arrête dès que la condition est fausse ; on dit alors qu'on « sort » de la boucle.

On commence par tester la condition ; si elle est vérifiée, on exécute le bloc d'instructions `blocWhile` (encore appelé le « corps » de la boucle) puis on reteste la condition : la condition est ainsi évaluée avant chaque exécution du corps de la boucle ; si la condition est à nouveau vérifiée on réexécute le bloc d'instructions `blocWhile` (on dit qu'on « repasse » dans la boucle) et ainsi de suite jusqu'à ce que la condition devienne fausse, auquel cas on « sort » de la boucle.

Définition 2.6 : itération conditionnelle
L'itération conditionnelle est une instruction de contrôle du flux d'instructions qui permet sous condition préalable de répéter zéro ou plusieurs fois la même instruction.

Fig. 2.15 : Boucle `while`
`while condition : blocWhile`



Exemple 2.10 : Table de multiplication

On cherche à écrire un algorithme qui affiche la table de multiplication d'un nombre n quelconque.

Exemple : $n = 9 \rightarrow$

```
1 * 9 = 9
2 * 9 = 18
3 * 9 = 27
4 * 9 = 36
5 * 9 = 45
6 * 9 = 54
7 * 9 = 63
8 * 9 = 72
9 * 9 = 81
```

L'affichage ci-contre est obtenu par l'algorithme suivant :

```
n = 9
i = 1
while i < 10:
    print(i, '*', n, '=', i*n)
    i = i + 1
```

L'algorithme précédent commence par initialiser `n` et le multiplicateur `i`. Ensuite, puisque `i < 10`, on rentre dans la boucle ; la première instruction `print` affiche successivement la valeur de `i`, une `*`, la valeur de `n`, le signe `=` puis la valeur du produit `i*n`, soit au premier passage : `1 * 9 = 9`. L'instruction suivante incrémente `i` qui devient ainsi égal à `2` (`1 + 1`). Les deux instructions du corps de la boucle `while` ayant été exécutées, on reteste la condition `i < 10` ; elle est à nouveau vérifiée puisque `i` vaut maintenant `2`. On repasse alors dans la boucle où on affiche `2 * 9 = 18` et où on incrémente `i` qui vaut maintenant `3` (`2 + 1`). On réitère ces opérations jusqu'à ce que `i` soit égal à `10` (`9 + 1`) ; entre-temps les 7 autres lignes de la table de multiplication par `9` ont été affichées. Lorsque `i` vaut `10`, la condition `i < 10` n'est plus vérifiée et on « sort » de la boucle `while`. ■ TD2.12

TD 2.12 : Dessin d'étoiles (1)
Ecrire un algorithme itératif qui affiche les n lignes suivantes (l'exemple est donné ici pour $n = 6$) :

```
******
****
***
**
*
```

Rappel Python *:*

```
>>> 5*'r'
'rrrrr'
>>> 2*'to'
'toto'
```

Dans une itération conditionnelle, la condition doit évoluer au cours des différents passages dans la boucle afin de pouvoir sortir de la boucle. C'est le cas de la boucle `while` de l'exemple 2.10 ci-dessus ; à chaque passage dans la boucle, le multiplicateur `i` est incrémenté : ainsi, partant de la valeur initiale `i = 1`, `i` deviendra nécessairement égal à `10` après 9 passages dans la boucle.

En ce qui concerne le nombre de passages dans la boucle, deux cas extrêmes peuvent se produire :

– la condition n'est pas vérifiée la première fois : on ne passe alors jamais dans la boucle.
  Exemple :
  ```
  x = 4
  y = 0
  while x < 0 : y = y + x
  ```
  x est positif ; la condition $x < 0$ n'est donc pas vérifiée la première fois : on ne rentre pas dans la boucle.
– la condition n'est jamais fausse : on ne sort jamais de la boucle ; on dit qu'on a affaire à une boucle « sans fin ».

Exemple :
```
x = 4
y = 0
while y >= 0 : y = y + x
```
y est initialement nul : on rentre dans la boucle ; l'instruction du corps de la boucle ne peut qu'incrémenter y puisque x est positif : y sera donc toujours positif et on ne sortira jamais de la boucle.

Le cas de la boucle « sans fin » est évidemment dû le plus souvent à une erreur involontaire qu'il faut savoir détecter assez vite pour éviter un programme qui « tourne » indéfiniment sans s'arrêter.

Remarque 2.13 : *Une erreur classique de l'apprenti informaticien est de ne pas faire évoluer la condition d'une boucle* `while` *: il « tombe » alors dans une boucle « sans fin » comme dans l'exemple ci-dessous :*

```
k = 1
p = x
while k < n :
  p = p*x
```
On entre dans la boucle ; on calcule la nouvelle valeur de `p` *puis on reteste la condition* `k <= n`*. Mais entre-temps* `k` *n'a pas évolué (il n'a pas été incrémenté) et donc la condition reste vraie et restera toujours vraie : l'exécution ne sortira plus de la boucle.*

TD 2.13 : Fonction factorielle

Ecrire un algorithme qui calcule $n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot (n-1) \cdot n$.

Exemple 2.11 : Fonction puissance

La puissance entière p *d'un nombre* x *est définie par :* $p = x^n = \prod_{k=1}^{n} x = \underbrace{x \cdot x \cdot x \cdots x}_{n \text{ fois}}$.

Pour calculer p « de tête », nous calculons successivement x, x^2, $x^3 \ldots$, x^{n-1} et x^n en mémorisant à chaque étape la puissance courante x^k et en multipliant simplement cette puissance par x pour obtenir la puissance immédiatement supérieure x^{k+1}. On s'arrête quand $k = n$: on a alors le résultat attendu ($p = x^n$). Cet algorithme peut s'écrire directement :

```
k = 1
p = x
while k < n:
    p = p*x
    k = k + 1
```
On commence par initialisé l'exposant k à 1 et la puissance p recherchée avec la valeur de $x^k = x^1 = x$ (`p = x`). Ensuite, pour chaque valeur de $k < n$, on multiplie la puissance courante p par x (`p*x`) qui devient la nouvelle puissance courante (`p = p*x`) et on n'oublie pas d'incrémenter l'exposant k ; ainsi à la fin de chaque itération, on a toujours $p = x^k \ \forall k \in [1; n]$.

■ TD2.13

Dans les exemples 2.10 et 2.11 précédents, on savait à l'avance combien de fois on devait passer dans les boucles : 9 fois pour afficher une table de multiplication et n fois pour calculer x^n. Mais ce n'est pas toujours le cas comme nous allons le constater dans les deux exemples suivants qui permettent de calculer respectivement la fonction exponentielle (exemple 2.12 : e^x) et le pgcd de 2 nombres (exemple 2.13 : $\mathrm{pgcd}(a, b)$).

Exemple 2.12 : Fonction exponentielle

La fonction exponentielle peut être calculée en fonction de son développement en série entière.

$$y = \exp(x) \approx \sum_{k=0}^{n} u_k = \sum_{k=0}^{n} \frac{x^k}{k!} = 1 + x + \frac{x^2}{2} + \ldots + \frac{x^n}{n!}$$

Les calculs seront arrêtés lorsque la valeur absolue du terme u_k *sera inférieure à un certain seuil* s $(0 < s < 1)$*. On n'utilisera ni la fonction* puissance (x^n) *ni la fonction* factorielle $(n!)$ *pour effectuer le calcul de* $\exp(x)$.

Pour ce calcul, on pourrait avoir la même démarche que dans l'exemple 2.11 de la puissance entière ; à savoir, on calcule le premier terme de la série ($x^0/0! = 1$), on le mémorise dans y, on calcule le 2ème terme ($x^1/1! = x$) et on l'ajoute à la valeur de y précédemment mémorisée ($1+x$), on calcule le 3ème terme ($x^2/2! = x^2/2$), on l'ajoute à y ($1 + x + x^2/2$) et ainsi de suite jusqu'à ce que le nouveau terme calculé vérifie la condition d'arrêt imposée ($|u_k| < s$). Mais chaque évaluation d'un nouveau terme fait intervenir *a priori* les fonctions puissance (x^k) et factorielle ($n!$)...qui sont très coûteuses en temps de calcul. On préfère remarquer que le terme u_{k+1} peut s'exprimer simplement en fonction du terme précédent u_k selon la relation de récurrence :

$$u_{k+1} = \frac{x^{k+1}}{(k+1)!} = \frac{x}{k+1} \cdot \frac{x^k}{k!} = \frac{x}{k+1} u_k$$

et qu'il est donc possible de mémoriser à chaque étape u_k pour calculer le terme suivant u_{k+1} sans utiliser ni la fonction puissance, ni la fonction factorielle. On obtient alors l'algorithme suivant :

```
k = 0
u = 1
y = u
while fabs(u) > s:
    u = u*x/(k+1)
    y = y + u
    k = k + 1
```

On initialise l'indice `k` à `0`, le terme `u` ($= u_k$) à la valeur du premier terme de la série ($x^0/0! = 1$) et `y` à ce premier terme (`y = u`). Puis, tant que la valeur absolue de u_k (`fabs(u)`) est supérieure au seuil `s`, on calcule le terme suivant u_{k+1} en utilisant la relation de récurrence obtenue ci-dessus (`u = u*x/(k+1)`) : le nouveau terme u_{k+1} est égal à l'ancien terme u_k multiplié par $x/(k+1)$; on ajoute u_{k+1} à la somme courante y (`y = y + u`) et on recommence sans oublier d'incrémenter `k` (`k = k + 1`). A la fin de chaque itération, on a toujours $y = \sum_{i=0}^{k} u_i$.

Ici, on connait la condition d'arrêt de la boucle ($|u_k| < s$) mais on ne sait pas *a priori* combien de fois on passera dans la boucle : on ne connait pas l'ordre n pour lequel on arrêtera le développement de la série entière. ■ TD2.14

TD 2.14 : Fonction sinus

Ecrire un algorithme qui calcule de manière itérative la fonction sinus en fonction de son développement en série entière.

$$\sin(x) \approx \sum_{k=0}^{n} u_k = \sum_{k=0}^{n} (-1)^k \frac{x^{2k+1}}{(2k+1)!}$$
$$= x - \frac{x^3}{6} + \frac{x^5}{120} + \ldots + (-1)^n \frac{x^{2n+1}}{(2n+1)!}$$

Les calculs seront arrêtés lorsque la valeur absolue du terme u_k sera inférieure à un certain seuil s ($0 < s < 1$). On n'utilisera ni la fonction puissance *(x^n) ni la fonction* factorielle *($n!$) pour effectuer le calcul de* $\sin(x)$.

Exemple 2.13 : Un calcul de pgcd (3)

On considère à nouveau la relation de récurrence qui caractérise le pgcd de 2 entiers a et b (voir exemples 2.3 et 2.9) : $\mathrm{pgcd}(a, b) = \mathrm{pgcd}(b, a\%b)$. *Cette relation nous dit de remplacer a par b et b par $r = a\%b$ autant de fois que possible jusqu'à ce que le reste soit nul* ($\mathrm{pgcd}(a, 0) = a$). *Ce qui conduit à l'algorithme suivant :*

```
while b != 0:
    r = a%b
    a = b
    b = r
```

A la fin de chaque passage dans le corps de la boucle, a, b et r prennent successivement les valeurs suivantes (r n'est pas connu avant la boucle) :

	a	*b*	*r*
avant la boucle	*12*	*18*	*?*
1er *passage*	*18*	*12*	*12*
2ème *passage*	*12*	*6*	*6*
3ème *passage*	*6*	*0*	*0*
après la boucle	*6*	*0*	*0*

Cet algorithme de calcul de pgcd est connu sous le nom d'algorithme d'Euclide (figure 2.16) et fait partie des grands classiques de l'algorithmique. ■TD2.15
Là encore, la condition d'arrêt est connue ($b \neq 0$) mais pas le nombre de passages dans la boucle. ■TD2.16

Dans tous les cas, que l'on connaisse ou non *a priori* le nombre de passages dans la boucle, on peut toujours utiliser l'itération conditionnelle (boucle `while`) pour répéter plusieurs fois un bloc d'instructions à condition de connaître la condition d'arrêt pour sortir de la boucle.

- Lorsqu'on connaît *a priori* le nombre de passages dans la boucle (voir exemples 2.10 et 2.11), il suffit de définir un compteur qui compte le nombre de fois où on passe dans la boucle : le multiplicateur `i` dans l'exemple de la table de multiplication et l'exposant `k` dans le calcul de la puissance. On initialise correctement ce compteur avant la boucle (`i = 1` ou `k = 1` selon l'exemple considéré), on incrémente le compteur dans la boucle (`i = i + 1` ou `k = k + 1`) et on sort de la boucle lorsque ce compteur dépasse le nombre de fois connu où on doit passer dans la boucle (`i < 10` ou `k <= n`).
- Lorsqu'on ne connaît pas *a priori* le nombre de passages dans la boucle (voir exemples 2.12 et 2.13), il faut absolument déterminer la condition d'arrêt de l'algorithme : $|u_k| < s$ pour le calcul de e^x et $b = 0$ dans l'exemple du pgcd. Il faut également s'assurer que cette condition sera bien atteinte au bout d'un certain nombre de passages dans la boucle : dans le calcul du pgcd par exemple, le reste r de la division $a \div b$ ne peut être qu'inférieur au diviseur b et comme b est remplacé par r dans le corps de la boucle, b ne peut que diminuer et atteindre 0 au bout du compte.

2.4.2 Parcours de séquences

Il est fréquent de manipuler des suites ordonnées d'éléments comme les chaînes de caractères (exemple : `s = "123"`), les tableaux (exemple : `t = [1,2,3]`) et les n-uplets (exemple : `u =`

Fig. 2.16 : Euclide
Euclide (né vers -325, mort vers -265) était un mathématicien de la Grèce antique, auteur des « Éléments », qui sont considérés comme l'un des textes fondateurs des mathématiques modernes. En particulier, le livre 7 traite de l'arithmétique : il y définit la division que l'on appelle division euclidienne et un algorithme pour calculer le plus grand commun diviseur de deux nombres, connu sous le nom d'algorithme d'Euclide.

TD 2.15 : Algorithme d'Euclide
Dans la tradition grecque, en comprenant un nombre entier comme une longueur, un couple d'entiers comme un rectangle, leur pgcd est la taille du plus grand carré permettant de carreler ce rectangle. L'algorithme décompose ce rectangle en carrés, de plus en plus petits, par divisions euclidiennes successives, de la longueur par la largeur, puis de la largeur par le reste, jusqu'à un reste nul. Faire la construction géométrique « à la grecque antique » qui permet de déterminer le pgcd d de $a = 21$ et $b = 15$ ($d = 3$).

TD 2.16 : Division entière
Ecrire un algorithme itératif qui calcule le quotient q et le reste r de la division entière $a \div b$ ($a = bq + r$).
On n'utilisera pas les opérateurs prédéfinis `/` et `%` mais on pourra s'inspirer du TD 2.4 page 45.

`1,2,3`). Chaque élément d'une séquence est accessible par son rang dans la séquence grâce à l'opérateur « crochets » : `sequence[rang]` (exemples : `s[1]`, `t[2]` ou `u[0]`) et par convention, le premier élément d'une séquence a le rang `0` (exemples : `s[1]` est le 2ème élément de la chaîne `s`, `t[2]` le 3ème élément du tableau `t` et `u[0]` le 1er élément du n-uplet `u`).

```
>>> s = "123"
>>> s[1]
'2'
```

```
>>> t = [1,2,3]
>>> t[2]
3
```

```
>>> u = 1,2,3
>>> u[0]
1
```

Remarque 2.14 : *Les éléments d'une chaîne de caractères sont eux-mêmes des chaînes de caractères à 1 seul caractère.*

```
>>> s = 'a4b2'
>>> s[1]
'4'
>>> s[3]
'2'
```

Définition 2.7 : SÉQUENCE

Une séquence est une suite ordonnée d'éléments, éventuellement vide, accessibles par leur rang dans la séquence.

Les principales opérations sur les séquences sont listées dans le tableau ci-dessous (d'après [10]).

Operation	Result
`x in s`	`True` if an item of `s` is equal to `x`, else `False`
`x not in s`	`False` if an item of `s` is equal to `x`, else `True`
`s1 + s2`	the concatenation of `s1` and `s2`
`s * n, n*s`	`n` copies of `s` concatenated
`s[i]`	i'th item of `s`, origin `0`
`s[i : j]`	
`s[i : j :step]`	Slice of `s` from `i` (included) to `j`(excluded). Optional `step` value, possibly negative (default : `1`).
`len(s)`	Length of s
`min(s)`	Smallest item of s
`max(s)`	Largest item of s
`range([start,] end [, step])`	Returns list of ints from `>=` `start` and `<` `end`. With 1 arg, list from `0..arg-1` With 2 args, list from `start..end-1` With 3 args, list from `start` up to `end` by `step`

La dernière fonction de ce tableau crée un tableau d'entiers compris entre `start` inclus (= 0 par défaut) et `end` exclus par pas de `step` (= 1 par défaut).

```
>>> range(3)
[0, 1, 2]
>>> range(3,9,2)
[3, 5, 7]
>>> range(7,0,-1)
[7, 6, 5, 4, 3, 2, 1]
```

```
>>> s = "bonjour"
>>> range(len(s))
[0, 1, 2, 3, 4, 5, 6]
>>> t = [4,2,6,5,3]
>>> range(max(t),min(t),-1)
[6, 5, 4, 3]
```

```
>>> u1 = 10,12,14
>>> u2 = 'a','b','c'
>>> range(len(u1+u2))
[0, 1, 2, 3, 4, 5]
>>> range(len(2*u2[0:2]))
[0, 1, 2, 3]
```

Exemple 2.14 : Affichage caractère par caractère

L'algorithme suivant affiche les caractères d'une chaîne `s`*, un par ligne :*

```
i = 0
while i < len(s) :
    print(s[i])
    i = i + 1
```

On se place au début de la chaîne (rang `i = 0`*) et, tant qu'on est à l'intérieur de la chaîne (*`i < len(s)`*), on affiche l'élément courant* `s[i]`*. Le rang* `i` *prend successivement les valeurs* `0`, `1`, `2` ... `len(s)-1` *(*`len(s)` *exclus).*

■TD2.17

L'affichage précédent nous a conduits à parcourir la chaîne `s` du premier élément (`i = 0`) au dernier élément (`i = len(s)-1`). On peut généraliser cet exemple au parcours d'une séquence `s` quelconque, du premier élément au dernier élément (parcours direct) :

```
i = 0
while i < len(s):
    # traiter s[i]
    i = i + 1
```

Se placer au début de la séquence : initialiser un entier `i` qui représentera le rang dans la séquence (rang initial : `i = 0`); puis tant qu'on est dans la séquence (condition : `0 <= i < len(s)`), traiter l'élément courant `s[i]` et passer à l'élément suivant (`i = i + 1`).

■TD2.18

Il existe une instruction de contrôle adaptée au parcours de séquence (figure 2.17) :

```
for element in sequence : blocFor
```

équivalente à :

```
i = 0
while i < len(s):
    element = sequence[i]
    blocFor
    i = i + 1
```

Ainsi, l'algorithme de l'exemple 2.14 ci-dessus peut se réécrire simplement sous la forme :

```
for element in s : print(element)
```

De même, l'algorithme de calcul de la fonction puissance (exemple 2.11) peut s'écrire avec une boucle `for` :

```
p = x
for i in range(n) : p = p*x
```

■TD2.19

TD 2.17 : Affichage inverse

Ecrire un algorithme qui affiche les caractères d'une chaîne `s`*, un par ligne en partant de la fin de la chaîne.*

TD 2.18 : Parcours inverse

Ecrire un algorithme qui parcourt en sens inverse une séquence `s` *quelconque (du dernier élément au premier élément).*

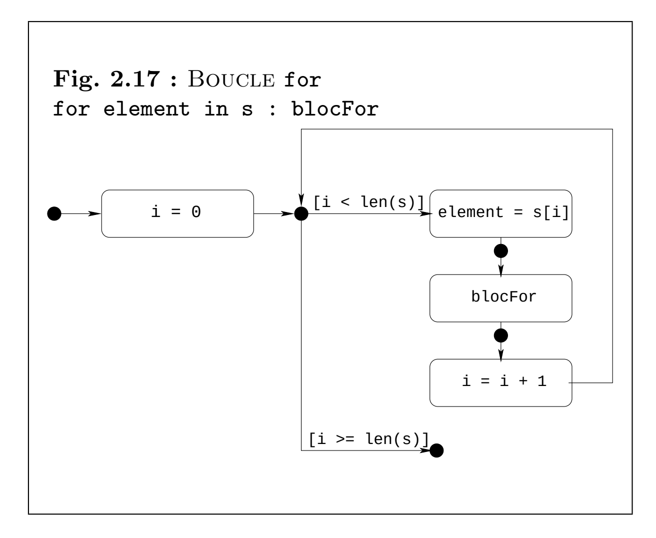


Fig. 2.17 : Boucle `for`
`for element in s : blocFor`

TD 2.19 : Suite arithmétique (2)

1. *Ecrire un algorithme qui calcule de manière itérative la somme $s = \sum_0^n u_k$ des n premiers termes d'une suite arithmétique $u_k = a + r \cdot k$. On utilisera une boucle* `for`.
2. *Comparer l'efficacité de cette approche itérative avec le calcul du TD 2.2 page 44.*

2.4.3 Imbrications de boucles

De la même manière que l'on peut cascader des alternatives simples (voir section 2.3.3), on peut encapsuler une boucle dans une autre boucle.

Exemple 2.15 : Tables de multiplication
Nous avons affiché une table de multiplication dans l'exemple 2.10 page 52. Nous voulons maintenant afficher les 9 premières tables de multiplication en réutilisant l'algorithme d'affichage d'une seule table. Il nous suffit pour cela de répéter 9 fois l'algorithme d'affichage d'une table en incrémentant le multiplicande n à chaque itération :

```
n = 1
while n <= 9:
    i = 1
    while i < 10:
        print(i, '*', n, '=', i*n)
        i = i + 1
    n = n + 1
```

On initialise `n` à `1` (on commence par la table de multiplication de `1`), puis on entre dans la boucle qui fera varier `n` de `1` à `9` (`while n <= 9`). On exécute l'algorithme d'affichage d'une table (exemple 2.10) et à la sortie de cet algorithme, on n'oublie pas d'incrémenter `n` (`n = n + 1`) pour passer à la table suivante. Et ainsi de suite jusqu'à la table de 9. Quand $n = 9$, son incrémentation lui affecte la valeur `10`, ce qui rend fausse la condition de la boucle (`n <= 9`) : on sort alors de la boucle extérieure.

■ TD2.20

TD 2.20 : Dessin d'étoiles (2)
Reprendre le TD 2.12 page 53 en supposant qu'on ne peut afficher qu'une étoile à la fois (on s'interdit ici la possibilité d'écrire `5*'*'` à la place de `'*****'` par exemple).

Cet exemple d'instruction composée pose explicitement le problème de la définition d'un bloc d'instructions : où commence et où termine un bloc d'instructions ? En effet, l'instruction `n = n + 1` fait-elle partie du bloc de la boucle intérieure (`while i < 10 :`) ou du bloc de la boucle extérieure (`while n <= 9 :`) ?

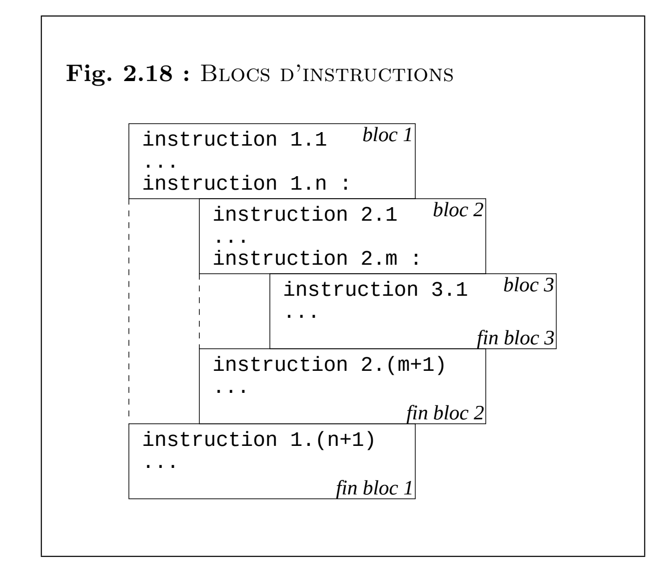
Les instructions composées ont toujours la même structure : une ligne d'en-tête terminée par un double point (:), suivie d'une ou de plusieurs instructions indentées (décalées à droite) sous cette ligne d'en-tête (figure 2.18).

```
ligne d'en-tête:
    première instruction du bloc
    ...
    dernière instruction du bloc
```

S'il y a plusieurs instructions indentées sous la ligne d'en-tête, elles doivent l'être exactement au même niveau (décalage de 4 caractères espace, par exemple). Ces instructions indentées constituent ce qu'on appellera désormais un bloc d'instructions. Un bloc d'instructions est une suite d'instructions formant un ensemble logique, qui n'est exécuté que dans certaines conditions définies dans la ligne d'en-tête. Dans l'exemple précédent, les deux lignes d'instructions indentées sous la ligne contenant l'instruction « `while i < 10 :` » constituent un même bloc logique :

Remarque 2.15 : *Dans le langage Pascal, les blocs d'instructions sont délimités par les mots-clés explicites `begin ... end`. En langage C, les blocs d'instructions sont délimités par des accolades (`{ ... }`). En Python, les blocs sont caractérisés par l'indentation identique de chaque instruction du bloc.*

Fig. 2.18 : Blocs d'instructions



ces deux lignes ne sont exécutées – toutes les deux – que si la condition testée avec l'instruction `while` est vérifiée, c'est-à-dire si le multiplicateur `i` est tel que `1 <= i < 10`.

Exemple 2.16 : Tables de vérité
Pour afficher les tables de vérité des opérateurs logiques de base (voir section 2.3.1) : négation (non, not, $\overline{a}$*), disjonction (ou,* or, $a+b$*) et conjonction (et,* and, $a \cdot b$*), on peut utiliser 2 boucles* `for` *imbriquées :*

Exemple : $a \cdot b$

a	b	$a \cdot b$
0	0	0
0	1	0
1	0	0
1	1	1

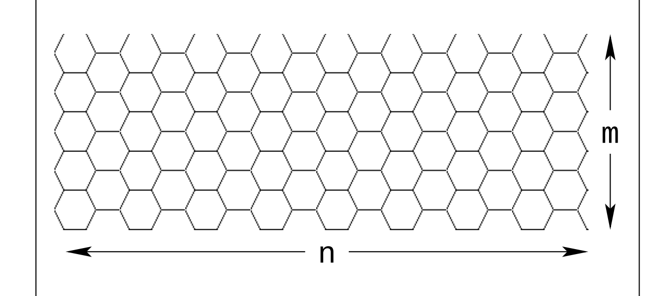
```
>>> for a in  [0,1]:
...   for b in [0,1]:
...     print(a, b, a and b)
...
0 0 0
0 1 0
1 0 0
1 1 1
```

■ TD2.21

TD 2.21 : Opérateurs booléens dérivés (2)
*A l'aide d'itérations imbriquées, afficher les tables de vérité des opérateurs logiques dérivés (voir TD 2.5) : ou exclusif (*xor, $a \oplus b$*), non ou (*nor, $\overline{a+b}$*), non et (*nand, $\overline{a \cdot b}$*), implication (*$a \Rightarrow b$*) et équivalence (*$a \Leftrightarrow b$*).*

Exemple 2.17 : Nid d'abeilles
Un motif en nid d'abeilles est formé de $n \times m$ *hexagones en quinconce comme sur la figure 2.19 ci-contre.*

Fig. 2.19 : Nid d'abeilles



Pour dessiner un tel motif, il faut d'abord savoir dessiner un hexagone de côté `a` en utilisant les instructions *à la* Logo de l'annexe 2.6.1 page 91 :

```
for k in range(6):
    forward(a)
    left(60)
```

puis une colonne de `m` hexagones de côté `a` à l'abscisse `x0` :

```
for j in range(m):
    y0 = a*sqrt(3)*(1/2. - j)
    up()
    goto(-x0,-y0)
    down()
    for k in range(6):
        forward(a)
        left(60)
```

et enfin `n` colonnes de `m` hexagones en quinconce :

```
for i in range(n):
    x0 = -3*i*a/2.
    for j in range(m):
        y0 = a*sqrt(3)*(1/2.*(i%2) - j)
        up()
        goto(-x0,-y0)
        down()
        for k in range(6):
            forward(a)
            left(60)
```

■ TD2.22

TD 2.22 : Damier
En utilisant les instructions à la Logo *de l'annexe 2.6.1 page 91, dessiner un damier rectangulaire de* $n \times m$ *cases.*

2.4.4 Exécutions de boucles

La maîtrise de l'algorithmique requiert deux qualités complémentaires [5] :

– il faut avoir une certaine intuition, car aucun algorithme ne permet de savoir *a priori* quelles instructions permettront d'obtenir le résultat recherché. C'est là qu'intervient la forme « d'intelligence » requise pour l'algorithmique : la « créativité » de l'informaticien. Il y a des gens qui possèdent au départ davantage cette intuition que les autres. Cependant, les réflexes, cela s'acquiert (en particulier, l'annexe 2.6.3 page 95 présente une méthode pour aider à construire des boucles). Et ce qu'on appelle l'intuition n'est finalement que de l'expérience accumulée, tellement répétée que le raisonnement, au départ laborieux, finit par devenir « spontané ».
– il faut être méthodique et rigoureux. En effet, chaque fois qu'on écrit une série d'instructions que l'on croit justes, il faut systématiquement se mettre mentalement à la place de la machine qui va les exécuter (sur papier ou dans sa tête) afin de vérifier si le résultat obtenu est bien celui que l'on voulait. Cette opération ne requiert pas d'intuition. Mais elle reste néanmoins indispensable si l'on ne veut pas écrire des algorithmes à l'« aveuglette ». Et petit à petit, à force de pratique, on fera de plus en plus souvent l'économie de cette dernière étape : l'expérience fera qu'on « verra » le résultat produit par les instructions, au fur et à mesure qu'on les écrira. Naturellement, cet apprentissage est long, et demande des heures de travail patient. Aussi, dans un premier temps, il faut éviter de sauter les étapes : la vérification méthodique, pas à pas, de chacun des algorithmes représente plus de la moitié du travail à accomplir. . .et le gage de progrès.

Remarque 2.16 : *Si en littérature « lire, c'est écrire dans sa tête », en algorithmique « lire un algorithme, c'est l'exécuter dans sa tête ».*

Pour améliorer la compréhension d'une boucle, on peut « tracer » son exécution de tête, à la main ou par programme. Dans tous les cas, l'idée est de suivre pas à pas l'évolution des

variables qui interviennent dans la boucle : on détermine leur valeur juste avant la boucle, à la fin de chaque itération et juste après la boucle. C'est ce qui a été fait dans l'exemple 2.13 page 55 du calcul du pgcd ou on a « pisté » les 3 variables concernées par ce calcul. ■TD2.23

TD 2.23 : Trace de la fonction factorielle
Tracer la fonction factorielle du TD 2.13 page 54.

Exemple 2.18 : Exécution d'une boucle

L'exécution pas à pas de l'algorithme ci-dessous donne le tableau de droite.

```
x = 2
n = 4
k = 1
p = x
while k < n:
    p = p*x
    k = k + 1
```

Les variables concernées par la boucle sont essentiellement `k` *et* `p` *(*`n` *et* `x` *ne varient pas au cours de l'algorithme) :*

	k	p
avant	*1*	*2*
pendant	*2*	*4*
pendant	*3*	*8*
pendant	*4*	*16*
après	*4*	*16*

TD 2.24 : Figure géométrique
Que dessinent les instructions suivantes ?

```
x0 = 0
y0 = 0
r = 10
n = 5
m = 10
for i in range(n) :
    up()
    y = y0 - 2*r*i
    x = x0 + r*(i%2)
    goto(x,y)
    for j in range(m) :
        down()
        circle(r)
        up()
        x = x + 2*r
        goto(x,y)
```

Exemple 2.19 : Nombres de Fibonacci

Les nombres de Fibonacci sont donnés par la suite $\begin{cases} f_0 = f_1 = 1 \\ f_n = f_{n-1} + f_{n-2} \ \forall n > 1 \end{cases}$

Les 10 premiers nombres de la suite de Fibonacci valent donc successivement $f_0 = 1$, $f_1 = 1$, $f_2 = 2$, $f_3 = 3$, $f_4 = 5$, $f_5 = 8$, $f_6 = 13$, $f_7 = 21$, $f_8 = 34$, *et* $f_9 = 55$.

Le nombre f_n $(n > 1)$ de Fibonacci se calcule selon l'algorithme itératif suivant :

```
f, f1, f2 = 2,1,1
for i in range(3,n+1) :
    f2 = f1
    f1 = f
    f = f1 + f2
```

On trace les 4 variables `i`, `f2`, `f1` et `f` concernées par la boucle dans le cas $n = 9$:

	i	f2	f1	f
avant	?	1	1	2
pendant	3	1	2	3
pendant	4	2	3	5
pendant	5	3	5	8
pendant	6	5	8	13
pendant	7	8	13	21
pendant	8	13	21	34
pendant	9	21	34	55
après	9	21	34	55

■TD2.24

2.5 Exercices complémentaires

2.5.1 Connaître

TD 2.25 : QCM (2)

(un seul item correct par question)

1. En Python, l'instruction « ne rien faire » se dit
   (a) `break`
   (b) `return`
   (c) `pass`
   (d) `continue`
2. Une variable informatique est un objet
   (a) équivalent à une variable mathématique
   (b) qui associe un nom à une valeur
   (c) qui varie nécessairement
   (d) qui modifie la mémoire
3. L'affectation consiste à
   (a) comparer la valeur d'une variable à une autre valeur
   (b) associer une valeur à une variable
   (c) incrémenter une variable
   (d) déplacer une variable en mémoire
4. Après la séquence

```
a = 13
b = 4
b = a
a = b
```

   les variables `a` et `b` sont telles que
   (a) `a = 13` et `b = 13`
   (b) `a = 4` et `b = 4`
   (c) `a = 4` et `b = 13`
   (d) `a = 13` et `b = 4`

5. Le résultat d'une comparaison est une valeur
   (a) réelle
   (b) qui dépend du type des arguments
   (c) booléenne
   (d) entière
6. Un opérateur booléen s'applique à des valeurs
   (a) booléennes
   (b) entières
   (c) réelles
   (d) alphanumériques
7. La fonction principale d'une instruction de test est
   (a) de passer d'instruction en instruction
   (b) de répéter une instruction sous condition
   (c) d'exécuter une instruction sous condition
   (d) d'interrompre l'exécution d'une instruction
8. Après la séquence

```
x = -3
if x < -4 : y = 0
elif x < -3 : y = 4 - x
elif x < -1 : y = x*x + 6*x + 8
elif x < 3 : y = 2 - x
else : y = -2
```

   la variable `y` est telle que
   (a) `y = -1`
   (b) `y = 0`
   (c) `y = 7`
   (d) `y = -2`
9. L'itération conditionnelle est une instruction de contrôle du flux d'instructions
   (a) qui permet d'exécuter une instruction sous condition préalable.
   (b) qui est vérifiée tout au long de son exécution.

(c) qui permet sous condition préalable de répéter zéro ou plusieurs fois la même instruction.

(d) qui permet de choisir entre plusieurs instructions.

10. On ne sort jamais d'une boucle si la condition d'arrêt
    (a) ne varie pas en cours d'exécution.
    (b) ne contient pas d'opérateurs booléens.
    (c) est toujours fausse.
    (d) n'est jamais fausse.
11. Que vaut `f` à la fin des instructions suivantes si $n = 5$?
```
f = 0
i = 1
while i < n+1:
    f = f + i
    i = i + 1
```
    (a) 6
    (b) 10
    (c) 15
    (d) 21
12. Une séquence est une suite ordonnée
    (a) d'éléments que l'on peut référencer par leur rang.
    (b) d'instructions formant un ensemble logique.
    (c) d'instructions conditionnelles.
    (d) de nombres
13. Dans la chaîne `s = 'gérard'`, `s[2]` vaut
    (a) `'é'`
    (b) `'r'`
    (c) `'gé'`
    (d) `'gér'`

14. Que vaut `f` à la fin des instructions suivantes si $n = 5$?

```
f = 1
for i in range(2,n+1) :
    f = f * i
```

    (a) 120
    (b) 720
    (c) 6
    (d) 24

15. Que vaut `f` à la fin des instructions suivantes si $n = 5$?

```
f, f1, f2 = 2,1,1
for i in range(3,n+1) :
    f2 = f1
    f1 = f
    f = f1 + f2
```

    (a) 3
    (b) 5
    (c) 8
    (d) 13

2.5.2 Comprendre

TD 2.26 : Unité de longueur

L'année-lumière (al) est une unité de distance utilisée en astronomie. Une année-lumière est la distance parcourue par un photon (ou plus simplement la lumière) dans le vide, en dehors de tout champ gravitationnel ou magnétique, en une année julienne (365,25 jours).

Ecrire une instruction qui permette de passer directement des années-lumière aux m/s sachant que la vitesse de la lumière dans le vide est de 299 792 458 m/s.

TD 2.27 : Permutation circulaire (2)

Effectuer une permutation circulaire gauche entre les valeurs de 3 entiers x, y et z.

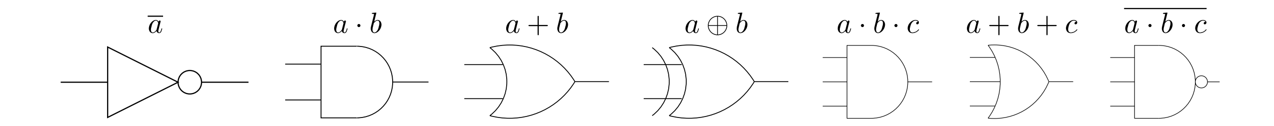
TD 2.28 : Séquence d'affectations (2)

Quelles sont les valeurs des variables n et s après la séquence d'affectations suivante ?

```
n = 1
s = n
n = n + 1
s = s + n
n = n + 1
s = s + n
n = n + 1
s = s + n
n = n + 1
s = s + n
```

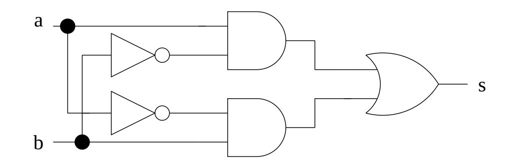
TD 2.29 : Circuits logiques (2)

On considère les conventions graphiques traditionnelles pour les opérateurs logiques :

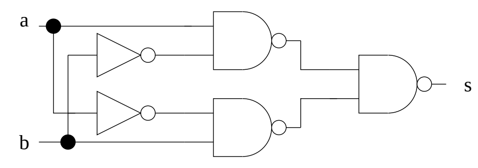


Donner les séquences d'affectations permettant de calculer la (ou les) sortie(s) des circuits logiques suivants en fonction de leurs entrées.

1. a et b sont les entrées, s la sortie.

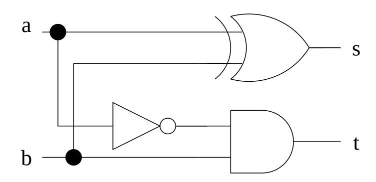


2. a et b sont les entrées, s la sortie.

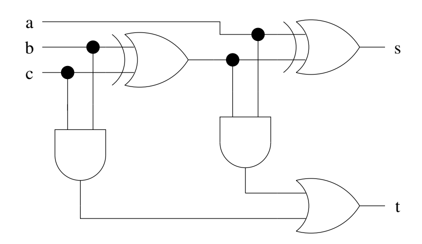




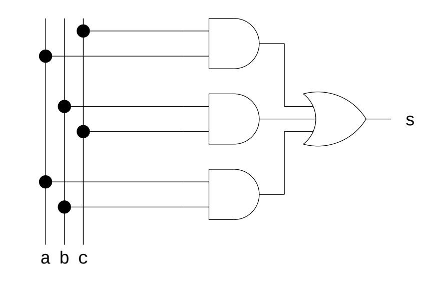
3. a et b sont les entrées, s et t les sorties.



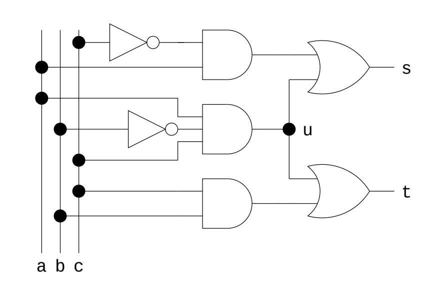
4. a, b et c sont les entrées, s et t les sorties.



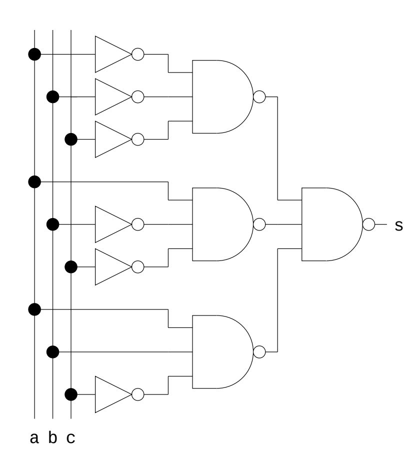
5. a, b et c sont les entrées et s la sortie.



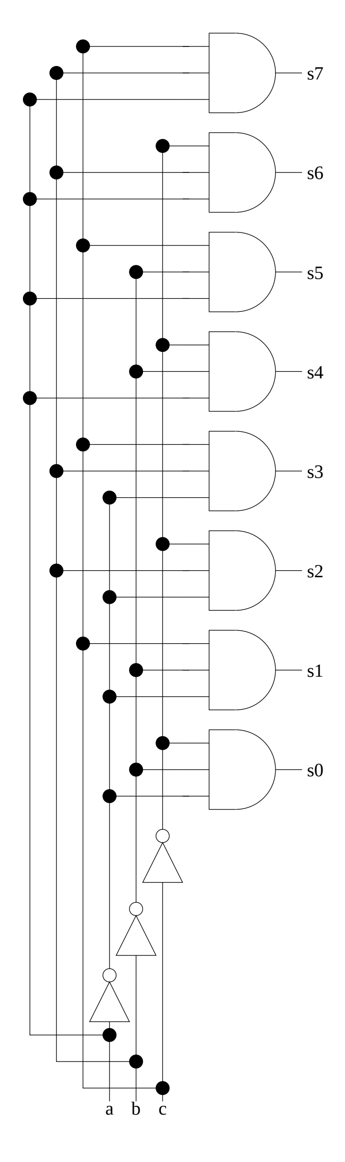
6. a, b et c sont les entrées, s et t les sorties.



7. a, b et c sont les entrées, s la sortie.



8. a, b et c sont les entrées, s_0,s_1...s_7 les sorties.



TD 2.30 : Alternative simple et test simple

Montrer à l'aide d'un contre-exemple que l'alternative simple de la figure 2.10 page 48 n'est pas équivalente à la séquence de tests simples suivante :

```
if condition : blocIf
if not condition : blocElse
```

TD 2.31 : Racines du trinome

Ecrire un algorithme qui calcule les racines x_1 et x_2 du trinome $ax^2 + bx + c$.

TD 2.32 : Séquences de tests

1. Quelle est la valeur de la variable x après la suite d'instructions suivante ?

```
x = -3
if x < 0 : x = -x
```

2. Quelle est la valeur de la variable y après la suite d'instructions suivante ?

```
x0 = 3
x = 5
if x < x0 : y = -1
else : y = 1
```

3. Quelle est la valeur de la variable y après la suite d'instructions suivante ?

```
p = 1
d = 0
r = 0
h = 1
z = 0
f = p and (d or r)
g = not r
m = not p and not z
g = g and (d or h or m)
if f or g : y = 1
else : y = 0
```

4. Quelle est la valeur de la variable ok après la suite d'instructions suivante ?

```
x = 2
y = 3
```

```
d = 5
h = 4
if x > 0 and x < d :
  if y > 0 and y < h : ok = 1
  else : ok = 0
else : ok = 0
```

5. Quelle est la valeur de la variable y après la suite d'instructions suivante ?

```
x = 3
y = -2
if x < y : y = y - x
elif x == y : y = 0
else : y = x - y
```

TD 2.33 : Racine carrée entière

Ecrire un algorithme qui calcule la racine carrée entière r d'un nombre entier positif n telle que $r^2 \leq n < (r+1)^2$.

TD 2.34 : Exécutions d'instructions itératives

1. Que fait cette suite d'instructions ?

```
x = 0
while x != 33 :
  x = input('entrer un nombre : ')
```

2. Que fait cette suite d'instructions ?

```
x = 0
while x <= 0 or x > 5 :
  x = input('entrer un nombre : ')
```

3. Que fait cette suite d'instructions ?

```
s = 0
for i in range(5) :
  x = input('entrer un nombre : ')
  s = s + x
```

4. Qu'affichent les itérations suivantes ?

```
for i in range(0,10) :
  for j in range(0,i) :
    print('*',end=' ')
  print()
```

5. Qu'affichent les itérations suivantes ?

```
for i in range(0,10) :
  j = 10 - i
  while j > 0 :
    print('*',end=' ')
    j = j - 1
  print()
```

6. Qu'affichent les itérations suivantes ?

```
for i in range(1,10) :
  for j in range(0,11) :
    print(i, 'x', j, ' = ', i*j)
  print()
```

7. Qu'affichent les itérations suivantes ?

```
for n in range(10) :
  for p in range(n+1) :
    num = 1
    den = 1
    for i in range(1,p+1) :
      num = num*(n-i+1)
      den = den*i
    c = num/den
    print(c,end=' ')
  print()
```

8. Qu'affichent les itérations suivantes ?

```
for n in range(0,15) :
  f = 1
  f1 = 1
  f2 = 1
  for i in range(2,n+1) :
    f = f1 + f2
    f2 = f1
    f1 = f
  print(f,end=' ')
```

9. Quelle est la valeur de la variable s à la fin des instructions suivantes ?

```
b = 2
k = 8
n = 23
s = 0
i = k - 1
q = n
while q != 0 and i >= 0 :
  s = s + (q%b)*b**(k-1-i)
  print(q%b,end=' ')
  q = q/b
  i = i - 1
```

2.5.3 Appliquer

TD 2.35 : Figures géométriques

1. Ecrire un algorithme qui calcule le périmètre p et la surface s d'un rectangle de longueur L et de largeur l.
2. Ecrire un algorithme qui calcule le périmètre p et la surface s d'un cercle de rayon r.
3. Ecrire un algorithme qui calcule la surface latérale s et le volume v d'un cylindre de rayon r et de hauteur h.

4. Ecrire un algorithme qui calcule la surface s et le volume v d'une sphère de rayon r.

TD 2.36 : Suites numériques

1. Ecrire un algorithme qui calcule la somme $s = \sum_0^n u_k$ des n premiers termes d'une suite arithmétique $u_k = a + bk$.
2. Ecrire un algorithme qui calcule la somme $s = \sum_0^n u_k$ des n premiers termes d'une suite géométrique $u_k = ab^k$.

TD 2.37 : Calcul vectoriel

1. Ecrire un algorithme qui calcule le module r et les cosinus directeurs a, b et c d'un vecteur de composantes (x, y, z).
2. Ecrire un algorithme qui calcule le produit scalaire p de 2 vecteurs de composantes respectives (x_1, y_1, z_1) et (x_2, y_2, z_2).
3. Ecrire un algorithme qui calcule les composantes (x_3, y_3, z_3) du produit vectoriel de 2 vecteurs de composantes respectives (x_1, y_1, z_1) et (x_2, y_2, z_2).
4. Ecrire un algorithme qui calcule le produit mixte v de 3 vecteurs de composantes respectives (x_1, y_1, z_1), (x_2, y_2, z_2) et (x_3, y_3, z_3).

TD 2.38 : Prix d'une photocopie

Ecrire un algorithme qui affiche le prix de n photocopies sachant que le reprographe facture 0,10 E les dix premières photocopies, 0,09 E les vingt suivantes et 0,08 E au-delà.

TD 2.39 : Calcul des impôts

Ecrire un algorithme qui affiche si un contribuable d'un pays imaginaire est imposable ou non sachant que :

- les hommes de plus de 18 ans paient l'impôt,
- les femmes paient l'impôt si elles ont entre 18 et 35 ans,
- les autres ne paient pas d'impôt.

TD 2.40 : Développements limités

Calculer chaque fonction ci-dessous en fonction de son développement en série entière $(\sum u_k)$. Les calculs seront arrêtés lorsque la valeur absolue du terme u_k sera inférieure à un certain seuil s $(0 < s < 1)$.
On n'utilisera ni la fonction *puissance* (x^n) ni la fonction *factorielle* $(n!)$.

1. $\sinh(x) \approx \sum_{k=0}^{n} \frac{x^{2k+1}}{(2k+1)!} = x + \frac{x^3}{6} + \frac{x^5}{120} + \ldots + \frac{x^{2n+1}}{(2n+1)!}$
2. $\cosh(x) \approx \sum_{k=0}^{n} \frac{x^{2k}}{(2k)!} = 1 + \frac{x^2}{2} + \frac{x^4}{24} + \ldots + \frac{x^{2n}}{(2n)!}$
3. $\cos(x) \approx \sum_{k=0}^{n} (-1)^k \frac{x^{2k}}{(2k)!} = 1 - \frac{x^2}{2} + \frac{x^4}{24} + \ldots + (-1)^n \frac{x^{2n}}{(2n)!}$
4. $\log(1+x) \approx \sum_{k=0}^{n} (-1)^k \frac{x^{k+1}}{k+1} = x - \frac{x^2}{2} + \frac{x^3}{3} + \ldots + (-1)^n \frac{x^{n+1}}{n+1}$, pour $-1 < x < 1$
5. $\arctan(x) \approx \sum_{k=0}^{n} (-1)^k \frac{x^{2k+1}}{(2k+1)} = x - \frac{x^3}{3} + \frac{x^5}{5} + \ldots + (-1)^n \frac{x^{2n+1}}{(2n+1)}$, pour $-1 < x < 1$

TD 2.41 : Tables de vérité

A l'aide d'itérations imbriquées, afficher les tables de vérité des circuits logiques du TD 2.29 page 67.

2.5.4 Analyser

TD 2.42 : Dessins géométriques

1. Que dessine la suite d'instructions suivante ?

```
forward(20)
right(144)
forward(20)
right(144)
forward(20)
right(144)
forward(20)
right(144)
forward(20)
right(144)
```

2. Que dessine la suite d'instructions suivante ?

```
forward(10)
left(45)
forward(10)
left(135)
forward(10)
left(45)
forward(10)
left(135)
```

TD 2.43 : Police d'assurance

Une compagnie d'assurance automobile propose 4 familles de tarifs du moins cher au plus onéreux : A, B, C et D. Le tarif dépend de la situation du conducteur.

- Un conducteur de moins de 25 ans et titulaire du permis depuis moins de deux ans, se voit attribuer le tarif D s'il n'a jamais été responsable d'accident. Sinon, la compagnie refuse de l'assurer.
- Un conducteur de moins de 25 ans et titulaire du permis depuis plus de deux ans, ou de plus de 25 ans mais titulaire du permis depuis moins de deux ans a le droit au tarif C s'il n'a jamais provoqué d'accident, au tarif D pour un accident, sinon il est refusé.
- Un conducteur de plus de 25 ans titulaire du permis depuis plus de deux ans bénéficie du tarif B s'il n'est à l'origine d'aucun accident et du tarif C pour un accident, du tarif D pour deux accidents, et refusé sinon.

Par ailleurs, pour encourager la fidélité de ses clients, la compagnie propose un contrat au tarif immédiatement inférieur s'il est assuré depuis plus d'un an.

Ecrire un algorithme qui propose un tarif d'assurance selon les caractéristiques d'un client potentiel.

TD 2.44 : Zéro d'une fonction

On recherche le zéro d'une fonction f continue sur un intervalle $[a, b]$ telle que $f(a).f(b) < 0$; il existe donc une racine de f dans $]a, b[$ que nous supposerons unique.

1. Ecrire un algorithme qui détermine le zéro de $\cos(x)$ dans $[1, 2]$ selon la méthode par dichotomie.

   Indications : on pose $x_1 = a$, $x2 = b$ et $x = (x_1 + x_2)/2$. Si $f(x1).f(x) < 0$, la racine est dans $]x_1, x[$ et on pose $x_2 = x$; sinon la racine est dans $]x, x_2[$ et on pose $x_1 = x$. Puis on réitère le procédé, la longueur de l'intervalle ayant été divisée par deux. Lorsque x_1 et x_2 seront suffisamment proches, on décidera que la racine est x.

2. Ecrire un algorithme qui détermine le zéro de $\cos(x)$ dans $[1, 2]$ selon la méthode des tangentes.

   Indications : soit x_n une approximation de la racine c recherchée : $f(c) = f(x_n) + (c - x_n)f'(x_n)$; comme $f(c) = 0$, on a : $c = x_n - f(x_n)/f'(x_n)$. Posons $x_{n+1} = x_n - f(x_n)/f'(x_n)$: on peut considérer que x_{n+1} est une meilleure approximation de c que x_n. On recommence le procédé avec x_{n+1} et ainsi de suite jusqu'à ce que $|x_{n+1} - x_n|$ soit inférieur à un certain seuil s.

3. Ecrire un algorithme qui détermine le zéro de $\cos(x)$ dans $[1, 2]$ selon la méthode des sécantes.

   Indications : reprendre la méthode des tangentes en effectuant l'approximation suivante : $f'(x_n) = (f(x_n) - f(x_{n-1}))/(x_n - x_{n-1})$.

4. Ecrire un algorithme qui détermine le zéro de $\cos(x)$ dans $[1, 2]$ selon la méthode des cordes.

   Indications : reprendre la méthode par dichotomie en prenant pour x le point d'intersection de la corde AB et de l'axe des abscisses : $x = (x_2 f(x_1) - x_1 f(x_2))/(f(x_1) - f(x_2))$, c'est-à-dire le point obtenu par la méthode des sécantes.

2.5.5 Solutions des exercices

TD 2.25 : QCM (2).

Les bonnes réponses sont extraites directement de ce document.

1c, 2b, 3c, 4a, 5c, 6a, 7d, 8a, 9d, 10d, 11c, 12a, 13b, 14a, 15b.

TD 2.26 : Unité de longueur.

$1\text{al} \approx 9.46 \cdot 10^{15}\text{m}$ et $1\text{m} \approx 1.06 \cdot 10^{-16}\text{al}$

```
>>> dAL = 1
>>> dM = dAL/(365.25*3600*24*299792458)
>>> dM
1.0570008340246154e-16
>>> dM = 1
>>> dAL = dM*365.25*3600*24*299792458
>>> dAL
9460730472580800.0
```

TD 2.27 : Permutation circulaire (2).

De manière classique, on passe par une variable intermédiaire `tmp` pour stocker la première valeur que l'on modifie.

```
>>> tmp = x
>>> x = y
>>> y = z
>>> z = tmp
```

En Python, on peut également directement écrire :

```
>>> x, y, z = y, z, x
```

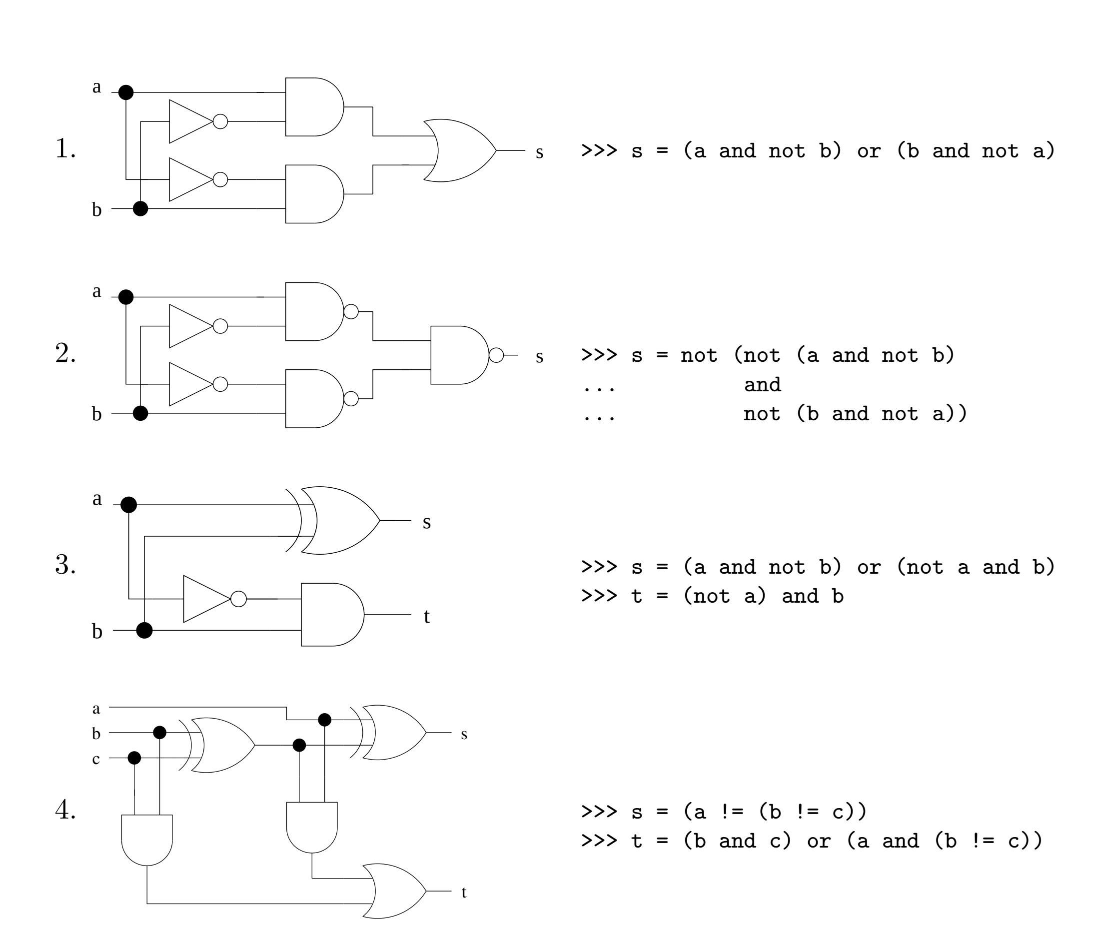
TD 2.28 : Séquence d'affectations (2).

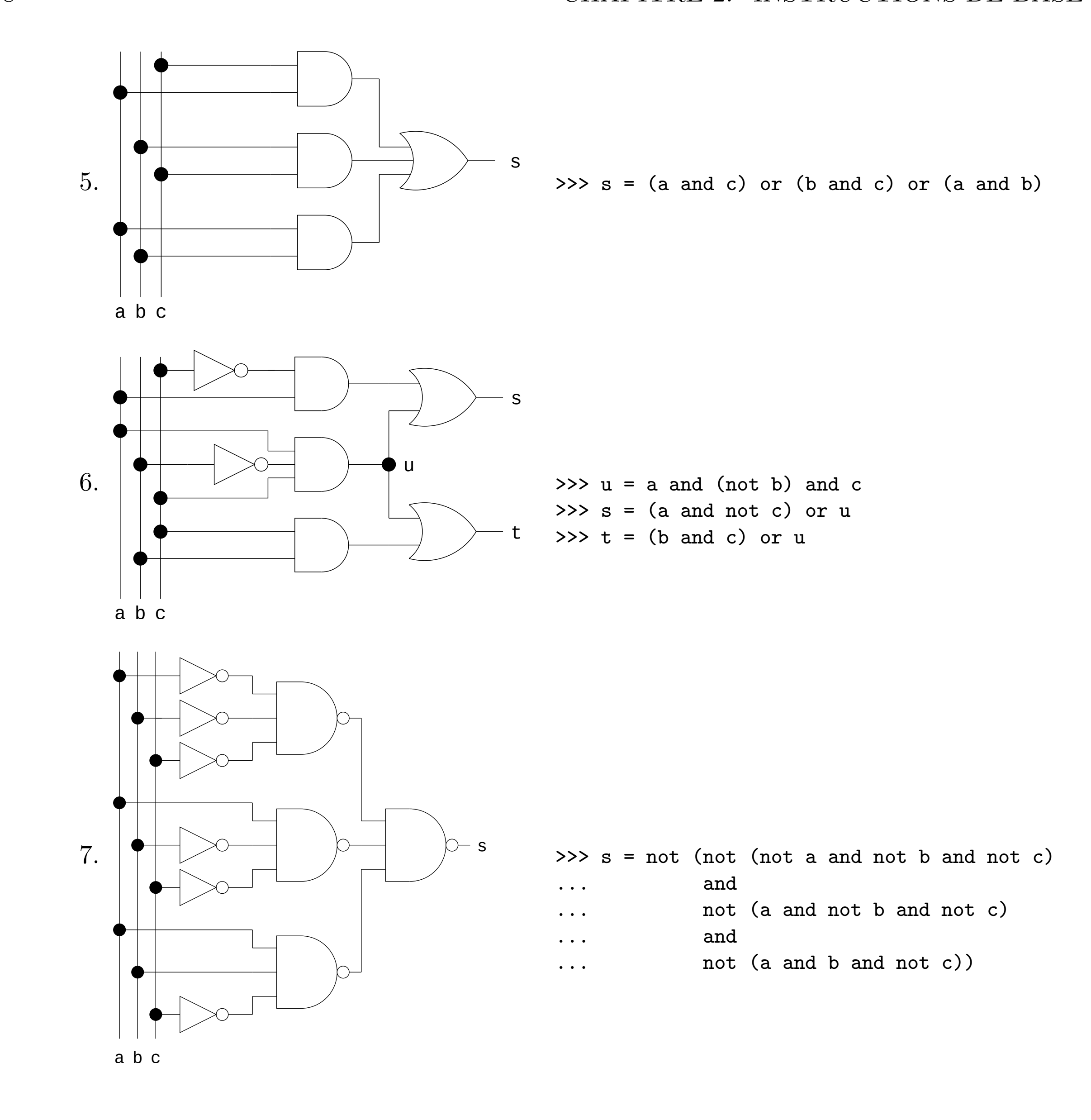
```
>>> n,s
(5, 15)
```

TD 2.29 : Circuits logiques (2).

Pour tester les différentes solutions obtenues, il faudra définir au préalable les entrées de chaque circuit logique. Par exemple :

```
>>> a = 1     # a = True
>>> b = 0     # b = False
>>> c = 1     # c = True
```





5.

```
>>> s = (a and c) or (b and c) or (a and b)
```

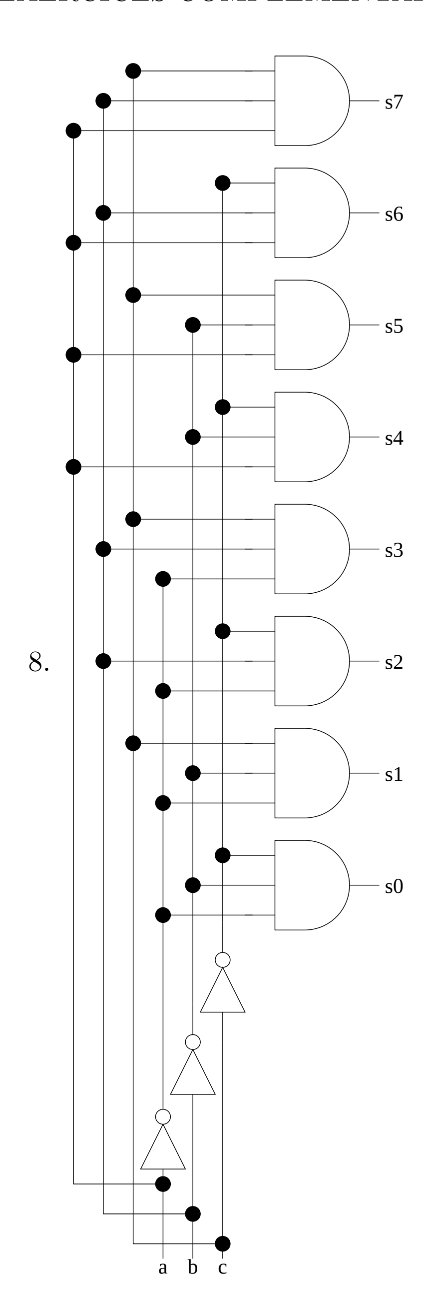
6.

```
>>> u = a and (not b) and c
>>> s = (a and not c) or u
>>> t = (b and c) or u
```

7.

```
>>> s = not (not (not a and not b and not c)
...           and
...           not (a and not b and not c)
...           and
...           not (a and b and not c))
```

8.




```
>>> s7 = a and b and c
>>> s6 = a and b and not c
>>> s5 = a and not b and c
>>> s4 = a and not b and not c
>>> s3 = not a and b and c
>>> s2 = not a and b and not c
>>> s1 = not a and not b and c
>>> s0 = not a and not b and not c
```

TD 2.30 : Alternative simple et test simple.

Il suffit que `blocIf` modifie la condition de telle manière qu'elle devienne fausse.

```
>>> x = - 1
>>> if x < 0 : x = 3
...
>>> if x >= 0 : x = -1
...
>>> print(x)
-1
```

```
>>> x = -1
>>> if x < 0 : x = 3
... else: x = -1
...
>>> print(x)
3
```

TD 2.31 : Racines du trinome.

```
>>> delta = b*b - 4*a*c
>>> if delta > 0 :
...   x1 = (-b - sqrt(delta))/(2*a)
...   x2 = (-b + sqrt(delta))/(2*a)
...   n = 2
... elif delta == 0 :
...   x1 = x2 = -b/(2*a)
...   n = 1
... else : n = 0
...
```

TD 2.32 : Séquences de tests.

1. ```
   >>> x
   3
   ```
2. ```
   >>> y
   1
   ```
3. ```
   >>> y
   1
   ```
4. ```
   >>> ok
   1
   ```
5. ```
   >>> y
   5
   ```

TD 2.33 : Racine carrée entière.

```
>>> n = 8
>>> r = 0
>>> while (r+1)**2 <= n :
...   r = r + 1
...
```
```

TD 2.34 : Exécutions d'instructions itératives.

1. L'algorithme demande d'entrer un nombre au clavier tant que ce nombre n'est pas égal à 33

```
...
entrer un nombre : 2
entrer un nombre : 0
entrer un nombre : 'fin'
entrer un nombre : 3.14
entrer un nombre : 33
>>>
```

2. L'algorithme demande d'entrer un nombre au clavier tant que ce nombre n'appartient pas à l'intervalle $]0;5]$.

```
...
entrer un nombre : 0
entrer un nombre : 6
entrer un nombre : 3.14
>>>
```

3. L'algorithme calcule la somme des 5 nombres entrés au clavier.

```
...
entrer un nombre : 1
entrer un nombre : 3
entrer un nombre : 2
entrer un nombre : 3
entrer un nombre : 6
>>> s
15
```

4. L'algorithme affiche des étoiles (*) selon la disposition suivante :

```
...

*
* *
* * *
* * * *
* * * * *
* * * * * *
* * * * * * *
* * * * * * * *
* * * * * * * * *
>>>
```

5. L'algorithme affiche des étoiles (*) selon la disposition suivante :

```
...
* * * * * * * * * *
* * * * * * * * *
* * * * * * * *
* * * * * * *
* * * * * *
* * * * *
* * * *
* * *
* *
*
>>>
```

6. L'algorithme affiche les tables de multiplication de 0 à 9.

```
...
1 x 0  =  0
1 x 1  =  1
1 x 2  =  2
1 x 3  =  3
1 x 4  =  4
1 x 5  =  5
1 x 6  =  6
1 x 7  =  7
1 x 8  =  8
1 x 9  =  9
1 x 10  =  10

2 x 0  =  0
2 x 1  =  2
2 x 2  =  4
2 x 3  =  6
2 x 4  =  8
2 x 5  =  10
2 x 6  =  12
2 x 7  =  14
2 x 8  =  16
2 x 9  =  18
2 x 10  =  20

3 x 0  =  0
3 x 1  =  3
3 x 2  =  6
3 x 3  =  9
3 x 4  =  12
3 x 5  =  15
3 x 6  =  18
etc...
```

7. L'algorithme affiche le triangle de Pascal jusqu'à l'ordre $n = 9$.
Il s'agit des coefficients du binôme $(x+y)^n = \sum_{p=0}^{n} \frac{n!}{p!(n-p)!} x^{n-p} y^p$

```
...
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1
1 6 15 20 15 6 1
1 7 21 35 35 21 7 1
1 8 28 56 70 56 28 8 1
1 9 36 84 126 126 84 36 9 1
>>>
```

8. L'algorithme affiche les 15 premiers nombres de la suite de Fibonacci : $u_0 = 1$, $u_1 = 1$, $u_n = u_{n-1} + u_{n-2}$.

```
...
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610
>>>
```

9. L'algorithme affiche les chiffres de la représentation de n sur k bits maximum en base b (du plus petit chiffre au plus grand). Après exécution, la valeur de s est simplement celle de n : on vérifie ainsi que la conversion en base b est correcte. Dans l'exemple ($n = 23$, $b = 2$ et $k = 8$), la valeur de s est $1{\cdot}2^0 + 1{\cdot}2^1 + 1{\cdot}2^2 + 0{\cdot}2^3 + 1{\cdot}2^4 = 23 = n$.

```
...
1 1 1 0 1
>>> s
23
```

TD 2.35 : Figures géométriques.

1. Périmètre p et surface s d'un rectangle de longueur L et de largeur l

```
>>> p = 2*(L+l)
>>> s = L*l
```

2. Périmètre p et surface s d'un cercle de rayon r

```
>>> p = 2*pi*r
>>> s = pi*r*r
```

3. Surface latérale s et volume v d'un cylindre de rayon r et de hauteur h

```
>>> s = 2*pi*r*h
>>> v = pi*r*r*h
```

4. Surface s et volume v d'une sphère de rayon r

```
>>> s = 4*pi*r*r
>>> v = 4*pi*r*r*r/3  # v = 4*pi*r**3/3
```

TD 2.36 : Suites numériques.

1. Somme $s = \sum_0^n u_k$ des n premiers termes d'une suite arithmétique $u_k = a + r \cdot k$

$$s = \sum_{k=0}^{n}(a+bk) = a(n+1) + b\sum_{k=1}^{n} k$$

avec $S = \sum_{k=1}^{n} k = (1+2+3+\cdots+n) = \dfrac{n(n+1)}{2}$

$$\begin{array}{rcccccccccccc} S & = & 1 & + & 2 & + & 3 & + & \cdots & + & n & & \\ S & = & n & + & n-1 & + & n-2 & + & \cdots & + & 1 & & \\ \hline 2S & = & (n+1) & + & (n+1) & + & (n+1) & + & \cdots & + & (n+1) & = & n(n+1) \end{array}$$

Version constante :

```
>>> s = a*(n+1) + b*n*(n+1)/2
```

Version itérative :

```
>>> s = 0
>>> for i in range(n+1) :
...     s = s + a + b*i
...
```

2. Somme $s = \sum_0^n u_k$ des n premiers termes d'une suite géométrique $u_k = a \cdot b^k$

$$s = \sum_{k=0}^{n} ab^k = a\sum_{k=0}^{n} b^k$$

où l'expression $N = (b^0 + b^1 + b^2 + \cdots + b^n)$ peut être vue comme le nombre $(111 \cdots 1)_b$ en base b. Or en base b, le nombre $(b-1)(b^0 + b^1 + b^2 + \cdots + b^n)$ est le nombre immédiatement inférieur à b^{n+1}, soit $(b-1)N = b^{n+1} - 1$.
Exemple en base $b = 10$: $999_{10} = 9(10^0 + 10^1 + 10^2) = 10^3 - 1$

$$S = \sum_{k=0}^{n} b^k = (b^0 + b^1 + b^2 + \cdots + b^n) = \frac{b^{n+1} - 1}{b - 1}$$

Version constante :

```
>>> s = a*(b**(n+1) - 1)/(b-1)
```

Version itérative :

```
>>> s = 0
>>> for i in range(n+1) :
...     s = s + a*b**i
...
```

TD 2.37 : Calcul vectoriel.

1. Module r et cosinus directeurs a, b et c d'un vecteur de composantes (x, y, z)

```
>>> r = sqrt(x*x + y*y + z*z)
>>> a1 = x/r
>>> a2 = y/r
>>> a3 = z/r
```

2. Le produit scalaire $p = \vec{a} \cdot \vec{b}$ de 2 vecteurs $\vec{a}$ et $\vec{b}$ est défini par $p = \sum_i a_i b_i$.

```
>>> p = x1*x2 + y1*y2 + z1*z2
```

3. Le produit vectoriel $\vec{c} = \vec{a} \times \vec{b}$ de 2 vecteurs $\vec{a}$ et $\vec{b}$ est un vecteur perpendiculaire au plan du parallélogramme défini par $\vec{a}$ et $\vec{b}$ et dont la longueur est égale à la surface de ce parallélogramme.

```
>>> x3 = y1*z2 - z1*y2
>>> y3 = z1*x2 - x1*z2
>>> z3 = x1*y2 - y1*x2
```

4. Le produit mixte $v = (\vec{a} \times \vec{b}) \cdot \vec{c}$ de 3 vecteurs $\vec{a}$, $\vec{b}$ et $\vec{c}$ représente le volume du parallélépipède construit sur ces 3 vecteurs.

```
>>> v = (y1*z2 - z1*y2)*x3 + (z1*x2 - x1*z2)*y3 + (x1*y2 - y1*x2)*z3
```

TD 2.38 : Prix d'une photocopie.

```
>>> if n > 30 : s = 10*0.1 + 20*0.09 + (n-30)*0.08
... elif n > 10 : s = 10*0.1 + (n - 10)*0.09
... else : s = n*0.1
...
```

TD 2.39 : Calcul des impôts.

```
>>> if a > 18 :
...   if s == 'm' : print('impôt')
...   elif a < 35 : print('impôt')
...   else : print("pas d'impôt")
... else : print("pas d'impôt")
...
```

TD 2.40 : Développements limités.

Pour chacun des algorithmes proposés, on pourra vérifier la valeur obtenue avec celle de la fonction correspondante dans le module `math` de Python.

```
>>> from math import *
```

1. $$y = \sinh(x) \approx \sum_{k=0}^{n} \frac{x^{2k+1}}{(2k+1)!} = x + \frac{x^3}{6} + \frac{x^5}{120} + \ldots + \frac{x^{2n+1}}{(2n+1)!}$$

   $$\text{avec } u_{k+1} = \frac{x^{2(k+1)+1}}{(2(k+1)+1)!} = \frac{x^2}{(2k+2)(2k+3)} \cdot \frac{x^{2k+1}}{(2k+1)!} = \frac{x^2}{(2k+2)(2k+3)} u_k$$

```
>>> k = 0
>>> u = x
>>> y = u
>>> s = 1.e-6
>>> while fabs(u) > s :
...   u = u*x*x/((2*k+2)*(2*k+3))
...   y = y + u
...   k = k + 1
...
```

2. $$y = \cosh(x) \approx \sum_{k=0}^{n} \frac{x^{2k}}{(2k)!} = 1 + \frac{x^2}{2} + \frac{x^4}{24} + \ldots + \frac{x^{2n}}{(2n)!}$$

   $$\text{avec } u_{k+1} = \frac{x^{2(k+1)}}{(2(k+1))!} = \frac{x^2}{(2k+1)(2k+2)} \cdot \frac{x^{2k}}{(2k)!} = \frac{x^2}{(2k+1)(2k+2)} u_k$$

```
>>> k = 0
>>> u = 1.
>>> y = u
>>> s = 1.e-6
>>> while fabs(u) > s :
...   u = u*x*x/((2*k+1)*(2*k+2))
...   y = y + u
...   k = k + 1
...
```

3. $y = \cos(x) \approx \sum_{k=0}^{n} (-1)^k \frac{x^{2k}}{(2k)!} = 1 - \frac{x^2}{2} + \frac{x^4}{24} + \ldots + (-1)^n \frac{x^{2n}}{(2n)!}$
avec $u_{k+1} = (-1)^k \frac{x^{2(k+1)}}{(2(k+1))!} = \frac{-x^2}{(2k+1)(2k+2)} \cdot (-1)^k \frac{x^{2k}}{(2k)!} = \frac{-x^2}{(2k+1)(2k+2)} u_k$

```
>>> k = 0
>>> u = 1.
>>> y = u
>>> s = 1.e-6
>>> while fabs(u) > s :
...   u = -u*x*x/((2*k+1)*(2*k+2))
...   y = y + u
...   k = k + 1
...
```

4. $y = \log(1+x) \approx \sum_{k=0}^{n} (-1)^k \frac{x^{k+1}}{k+1} = x - \frac{x^2}{2} + \frac{x^3}{3} + \ldots + (-1)^n \frac{x^{n+1}}{n+1}$, pour $-1 < x < 1$
avec $u_{k+1} = (-1)^k \frac{x^{(k+1)+1}}{(k+1)+1} = -x \frac{k+1}{k+2} \cdot (-1)^k \frac{x^{k+1}}{k+1} = -x \frac{k+1}{k+2} u_k$

```
>>> if fabs(x) < 1 :
...   k = 0
...   u = x
...   y = u
...   s = 1.e-6
...   while fabs(u) > s :
...     u = -u*x*(k+1)/(k+2)
...     y = y + u
...     k = k + 1
...
```

5. $y = \arctan(x) \approx \sum_{k=0}^{n} (-1)^k \frac{x^{2k+1}}{(2k+1)} = x - \frac{x^3}{3} + \frac{x^5}{5} + \ldots + (-1)^n \frac{x^{2n+1}}{(2n+1)}$, pour $-1 < x < 1$
avec $u_{k+1} = (-1)^k \frac{x^{2(k+1)+1}}{(2(k+1)+1)} = -x^2 \frac{2k+1}{2k+3} \cdot (-1)^k \frac{x^{2k+1}}{(2k+1)} = -x^2 \frac{2k+1}{2k+3} u_k$

```
>>> if fabs(x) < 1 :
...   k = 0
...   u = x
...   y = u
...   s = 1.e-6
...   while fabs(u) > s :
...     u = -u*x*x*(2*k+1)/(2*k+3)
...     y = y + u
...     k = k + 1
...
```

TD 2.41 : Tables de vérité.

1.
```
>>> for a in [0,1] :
...   for b in [0,1] :
...     for c in [0,1] :
...       s = (a and not b) or
...           (b and not a)
...       print(a, b, c, s)
...
0 0 0 0
0 0 1 0
0 1 0 1
0 1 1 1
1 0 0 1
1 0 1 1
1 1 0 0
1 1 1 0
```

2.
```
>>> for a in [0,1] :
...   for b in [0,1] :
...     s = not (not (a and not b)
...         and
...         not (b and not a))
...     print(a, b, s)
...
0 0 0
0 1 1
1 0 1
1 1 0
```

3.
```
>>> for a in [0,1] :
...   for b in [0,1] :
...     s = (a and not b) or (not a and b)
...     t = (not a) and b
...     print(a, b, s, t)
...
0 0 0 0
0 1 1 1
1 0 1 0
1 1 0 0
```

4.
```
>>> for a in [0,1] :
...   for b in [0,1] :
...     for c in [0,1] :
...       s = (a != (b != c))
...       t = (b and c) or (a and (b != c))
...       print(a, b, c, s, t)
...
```

5.
```
>>> for a in [0,1] :
...   for b in [0,1] :
...     for c in [0,1] :
...       s = not (not (a and b) and
...           not (a and c) and
...           not (b and c))
...       print(a, b, c, s)
...
0 0 0 0
0 0 1 0
0 1 0 0
0 1 1 1
1 0 0 0
1 0 1 1
1 1 0 1
1 1 1 1
```

6.
```
>>> for a in [0,1] :
...   for b in [0,1] :
...     for c in [0,1] :
...       s = not (not (not a and
...                     not b and not c) and
...           not (a and not b and
...                not c) and
...           not (a and b and not c))
...       print(a, b, c, s)
...
0 0 0 1
0 0 1 0
0 1 0 0
0 1 1 0
1 0 0 1
1 0 1 0
1 1 0 1
1 1 1 0
```

7.
```
>>> for a in [0,1] :
...   for b in [0,1] :
...     for c in [0,1] :
...       u = (b and not c) or (not b and c)
...       s = (a and not u) or (not a and u)
...       t = (b and c) or (a and u)
...       print(a, b, c, u, s, t)
...
0 0 0 0 0 0
0 0 1 1 1 0
0 1 0 1 1 0
0 1 1 0 0 1
1 0 0 0 1 0
1 0 1 1 0 1
1 1 0 1 0 1
1 1 1 0 1 1
```

8.
```
>>> for a in [0,1] :
...   for b in [0,1] :
...     for c in [0,1] :
...       s7 = a and b and c
...       s6 = a and b and not c
...       s5 = a and not b and c
...       s4 = a and not b and not c
...       s3 = not a and b and c
...       s2 = not a and b and not c
...       s1 = not a and not b and c
...       s0 = not a and not b and not c
...       print(a, b, c, s0, s1, s2, s3, s4, s5, s6, s7)
...
0 0 0 1 0 0 0 0 0 0 0
0 0 1 0 1 0 0 0 0 0 0
0 1 0 0 0 1 0 0 0 0 0
0 1 1 0 0 0 1 0 0 0 0
1 0 0 0 0 0 0 1 0 0 0
1 0 1 0 0 0 0 0 1 0 0
1 1 0 0 0 0 0 0 0 1 0
1 1 1 0 0 0 0 0 0 0 1
```

TD 2.42 : Dessins géométriques.

1. Une étoile à 5 branches

2. Un losange

TD 2.43 : Police d'assurance.

Il s'agit en fait d'une assurance à points (p) qui dépend de l'âge a du conducteur, de l'ancienneté d du permis, du nombre n d'accidents et du nombre r d'années en tant qu'assuré fidèle.

```
>>> p = 0
>>> if a < 25 : p = p + 1            # jeune
...
>>> if d < 2 : p = p + 1             # jeune conducteur
...
>>> p = p + n                        # accidents
>>>
>>> if p < 3 and r > 1 : p = p - 1   # fidélité
...
>>> if p == -1 : print('tarif A')
... elif p == 0 : print('tarif B')
... elif p == 1 : print('tarif C')
... elif p == 2 : print('tarif D')
... else : print('refus d'assurer')
...
```

TD 2.44 : Zéro d'une fonction.

1. Méthode par dichotomie

```
>>> x1 = 1.
>>> x2 = 3.
>>> x = (x1 + x2)/2.
>>> s = 1.e-9
>>> f = cos
>>> while (x2 - x1) > s :
...   if f(x1)*f(x) < 0 : x2 = x
...   else : x1 = x
...   x = (x1 + x2)/2.
>>> x, cos(x)
(1.5707963271997869, -4.048902822446996e-10)
```

2. Méthode des tangentes

```
>>> x1 = 1.
>>> x2 = 2.
>>> s = 1.e-9
>>> f = cos
>>> df = sin
>>> x = x2 - f(x2)/(-df(x2))
>>> while fabs(x-x2) > s :
...   x2 = x
...   x = x - f(x)/(-df(x))
...
>>> x, cos(x)
(1.5707963267948966, 6.1230317691118863e-17)
```

3. Méthode des sécantes

```
>>> x1 = 1.
>>> x2 = 2.
>>> s = 1.e-9
>>> f = cos
>>> df = (f(x2)-f(x1))/(x2-x1)
>>> x = x2 - f(x2)/df
>>> while fabs(x-x2) > s :
...   x2 = x
...   df = (f(x2)-f(x1))/(x2-x1)
...   x = x - f(x)/df
...
>>> x, cos(x)
(1.570796326805755, -1.085836403972619e-11)
```

4. Méthode des cordes

```
>>> x1 = 1.
>>> x2 = 3.
>>> x = (x2*f(x1) - x1*f(x2))/(f(x1)-f(x2))
>>> s = 1.e-9
>>> f = cos
>>> while (x2-x1) > s :
...   if f(x1)*f(x) < 0 : x2 = x
...   else : x1 = x
...   x = (x2*f(x1) - x1*f(x2))/(f(x1)-f(x2))
...
>>> x, cos(x)
(1.5707963267948966, 6.1230317691118863e-17)
```

2.6 Annexes

2.6.1 Instructions Logo

Logo is the name for a philosophy of education and a continually evolving family of programming languages that aid in its realization (Harold Abelson, Apple Logo, 1982). *This statement sums up two fundamental aspects of Logo and puts them in the proper order. The Logo programming environments that have been developed over the past 28 years are rooted in constructivist educational philosophy, and are designed to support constructive learning. [...] Constructivism views knowledge as being created by learners in their own minds through interaction with other people and the world around them. This theory is most closely associated with Jean Piaget, the Swiss psychologist, who spent decades studying and documenting the learning processes of young children.*

Logo Foundation : `http://el.media.mit.edu/logo-foundation`

On suppose connues les procédures de tracés géométriques *à la* Logo :

`degrees()` fixe l'unité d'angle en degrés

`radians()` fixe l'unité d'angle en radians

`reset()` efface l'écran et réinitialise les variables

`clear()` efface l'écran

`up()` lève le crayon

`down()` abaisse le crayon

`forward(d)` avance d'une distance d

`backward(d)` recule d'une distance d

`left(a)` tourne sur la gauche d'un angle a

`right(a)` tourne sur la droite d'un angle a

`goto(x,y)` déplace le crayon à la position (x, y)

`towards(x,y)` oriente vers le point de coordonnées (x, y)

`setheading(a)` oriente d'un angle a par rapport à l'axe des x

`position()` donne la position (x, y) du crayon

`heading()` donne l'orientation a du déplacement

`circle(r)` trace un cercle de rayon r

`circle(r,a)` trace un arc de cercle de rayon r et d'angle au sommet a.

2.6.2 Instructions Python

Les principales instructions Python sont listées dans les tableaux ci-dessous, tableaux extraits du Python *2.5 Quick Reference Guide* [10].

Miscellaneous statements

Statement	**Result**
`pass`	Null statement
`del name[, name]*`	Unbind `name`(s) from object
`print([s1 [, s2 ]*)`	Writes to `sys.stdout`, or to `file-object` if supplied. Puts spaces between arguments `si`. Puts newline at end unless arguments end with `end=` (ie : `end=' '`). `print` is not required when running interactively, simply typing an expression will print its value, unless the value is `None`.
`input([prompt])`	Prints `prompt` if given. Reads input and evaluates it.

Assignment operators

Operator	**Result**
`a = b`	Basic assignment - assign object `b` to label `a`
`a += b`	`a = a + b`
`a -= b`	`a = a - b`
`a *= b`	`a = a * b`
`a /= b`	`a = a / b`
`a //= b`	`a = a // b`
`a %= b`	`to a = a % b`
`a **= b`	`to a = a ** b`
`a &= b`	`a = a & b`
`a |= b`	`a = a | b`
`a ^= b`	`a = a ^ b`
`a >>= b`	`a = a >> b`
`a <<= b`	`a = a << b`

Control flow statements

Statement	Result

`if condition :`
`  suite`
`[elif condition : suite]*`
`[else :`
`  suite]`
	Usual `if/else if/else` statement.

`while condition :`
`  suite`
`[else :`
`  suite]`
	Usual `while` statement. The `else suite` is executed after loop exits, unless the loop is exited with `break`.

`for element in sequence :`
`  suite`
`[else :`
`  suite]`
	Iterates over `sequence`, assigning each element to `element`. Use built-in `range` function to iterate a number of times. The `else suite` is executed at end unless loop exited with `break`.

`break` `continue`	Immediately exits for or while loop. Immediately does next iteration of for or while loop.
`return [result]`	Exits from function (or method) and returns `result` (use a tuple to return more than one value).

Name space statements

Imported module files must be located in a directory listed in the `path` (`sys.path`).

Packages : a package is a name space which maps to a directory including module(s) and the special initialization module `__init__.py` (possibly empty).

Packages/directories can be nested. You address a module's symbol via `[package.[package...].module.symbol` .

Statement	**Result**

`import module1 [as name1] [, module2]*`	Imports modules. Members of module must be referred to by qualifying with `[package.]module` name, e.g. :
	`import sys ; print(sys.argv)`
	`import package1.subpackage.module`
	`package1.subpackage.module.foo()`
	`module1` renamed as `name1`, if supplied.

`from module import name1 [as othername1][, name2]*`	Imports `names` from module `module` in current namespace, e.g. :
	`from sys import argv ; print(argv)`
	`from package1 import module ; module.foo()`
	`from package1.module import foo ; foo()`
	`name1` renamed as `othername1`, if supplied.

`from module import *`	Imports all names in `module`, except those starting with _.

2.6.3 Construction d'une boucle

Un algorithme est un mécanisme qui fait passer un « système » d'une « situation » dite initiale (ou précondition) à une « situation » finale (postcondition ou but). Le couple (situation initiale, situation finale) est appelé spécification de l'algorithme. L'algorithmique vise donc à construire rationnellement des algorithmes à partir de leur spécification.

Exemple 2.20 : Enfoncer un clou
On dispose d'une planche, d'un marteau et d'un clou (situation initiale) et on veut que le clou soit enfoncé dans la planche jusqu'à la tête (situation finale).

Le travail à réaliser consiste donc à planter légèrement le clou à la main de façon qu'il tienne seul, puis à taper sur la tête du clou avec le marteau tant que la tête ne touche pas la planche. Le nombre de coups nécessaire est a priori *inconnu.*

Le raisonnement qui permet de passer d'une compréhension intuitive d'un tel énoncé à l'algorithme n'est pas toujours facile à concevoir d'un coup. Dans le cas d'une boucle on pourra systématiser la conception de l'algorithme autour de 4 étapes (d'après [7] et [11]) :

1. **Invariant** (ou hypothèse de récurrence) : « Le clou est planté dans la planche ».
2. **Condition d'arrêt** : « La tête touche la planche ».
3. **Progression** : « Frapper un coup de marteau de façon à enfoncer un peu plus le clou ».
4. **Initialisation** : « Planter légèrement le clou à la main ».

Il faut noter que les étapes 1 et 2 définissent des situations tandis que les étapes 3 et 4 concernent des actions. Dans cette section, on notera les situations entre crochets (`[]`) pour les distinguer des actions.

- La recherche d'un invariant est l'étape clé autour de laquelle s'articule la conception des boucles. La conjonction de l'invariant et de la condition d'arrêt conduit logiquement au but recherché :

  `[« invariant » and « condition d'arrêt »] ⇒ [« postcondition »]`

  La condition d'arrêt seule n'implique pas le but : un clou posé sur la planche la pointe en l'air a bien la tête qui touche la planche, mais il n'est pas planté dans la planche.

- La progression doit :
  - conserver l'invariant (pas de coup de marteau qui déplanterait le clou !). Plus précisément, la progression est un fragment d'algorithme défini par les situations initiale et finale suivantes :

    situation initiale : `[« invariant » and not « condition d'arrêt »]`
    situation finale : `[« invariant »]`

    Dans l'exemple du clou, étant donnée la précondition `[« le clou est planté dans la planche » and « la tête ne touche pas la planche »]`, et la postcondition `[« le clou est enfoncé dans la planche »]`, une solution à la progression est « `frapper un coup de marteau` ».

```
[« invariant » and not « condition d'arrêt »]
« progression »
[« invariant »]
```

  - faire effectivement progresser vers le but pour faire en sorte que la condition d'arrêt soit atteinte au bout d'un temps fini. Ici il est nécessaire de faire décroître la hauteur du clou au dessus de la planche.
- L'initialisation doit instaurer l'invariant. Plus précisément, elle doit, partant de la précondition, atteindre l'invariant.

```
[« précondition »]
« initialisation »
[« invariant »]
```

Pour enfoncer un clou dans une planche, on exécutera ainsi l'algorithme suivant :

```
[« on dispose d'une planche d'un marteau, d'un clou »]
« Planter légèrement le clou à la main »
[« le clou est planté dans la planche »]
while [« la tête du clou ne touche pas la planche »] :
    « frapper un coup de marteau sur la tête du clou »
    [« le clou est planté dans la planche »]
[« le clou est enfoncé dans la planche »]
```

D'une manière plus générale, les 4 étapes de construction d'une boucle sont les suivantes :

1. **Invariant :** proposer une situation générale décrivant le problème posé (hypothèse de récurrence). C'est cette étape qui est la plus délicate car elle exige de faire preuve d'imagination.

   Exemple de la puissance 2.11 :
   $x^{k+1} = x \cdot x^k$

   Exemple du pgcd 2.13 :
   $\text{pgcd}(a, b) = \text{pgcd}(b, a \bmod b)$

2. **Condition d'arrêt :** à partir de la situation générale imaginée en [1], on doit formuler la condition qui permet d'affirmer que l'algorithme a terminé son travail. La situation dans laquelle il se trouve alors est appelée situation finale. La condition d'arrêt fait sortir de la boucle.

   Exemple de la puissance 2.11 :
   `k > n`

   Exemple du pgcd 2.13 :
   `b == 0`

3. **Progression :** se « rapprocher » de la situation finale, tout en faisant le nécessaire pour conserver à chaque étape une situation générale analogue à celle retenue en [1]. La progression conserve l'invariant.

   Exemple de la puissance 2.11 :
   ```
   p = p*x
   k = k + 1
   ```

   Exemple du pgcd 2.13 :
   ```
   r = a%b
   a = b
   b = r
   ```

4. **Initialisation :** initialiser les variables introduites dans l'invariant pour que celui-ci soit vérifié avant d'entrer dans la boucle. L'initialisation « instaure » l'invariant.

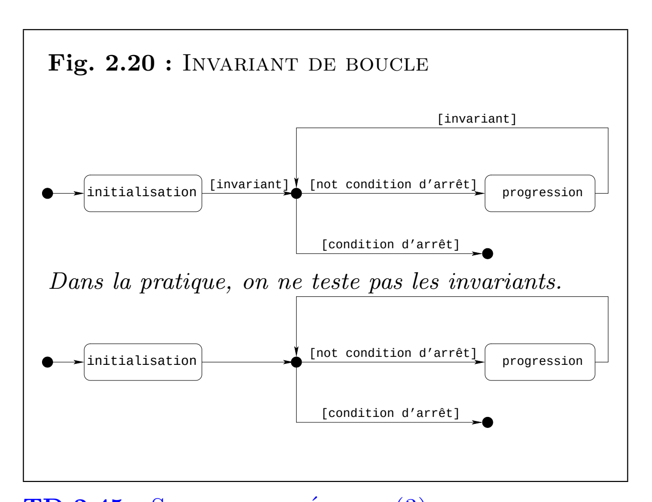
   Exemple de la puissance 2.11 :
   ```
   k = 1
   p = x
   ```

   Exemple du pgcd 2.13 :
   —

5. **Boucle finale :** Une fois les 4 étapes précédentes menées à leur terme, l'algorithme recherché aura la structure finale suivante (figure 2.20) : ■TD2.45

```
[« précondition »]
« initialisation »
[« invariant »]
while not [« condition d'arrêt »] :
    « progression »
    [« invariant »]
[« postcondition »]
```

Fig. 2.20 : Invariant de boucle



TD 2.45 : Suite arithmétique (3)
Reprendre le TD 2.19 page 58 en explicitant l'invariant, la condition d'arrêt, la progression et l'initialisation de la boucle retenue.

Quand on sort de la boucle, la situation finale attendue est atteinte.
Dans la pratique, on ne garde que les instructions :

```
« initialisation »
while [not « condition d'arrêt »] :
    « progression »
```

Exemple de la puissance 2.11 :

```
k = 1
p = x
while not k > n :
    p = p*x
    k = k + 1
```

Exemple du pgcd 2.13 :

```
while not b == 0 :
    r = a%b
    a = b
    b = r
```

Un des problèmes, pour l'apprenti informaticien, est que la boucle finale ainsi obtenue ne fait pas apparaître explicitement l'invariant dans le code. L'invariant est une aide conceptuelle pour construire la boucle, mais pas pour l'exécuter.

Définition 2.8 :
Un invariant de boucle est une propriété vérifiée tout au long de l'exécution de la boucle.

Cette façon de procéder permet de « prouver » la validité de l'algorithme au fur et à mesure de son élaboration. En effet la situation générale choisie en [1] est en fait l'invariant qui caractérise la boucle `while`. Cette situation est satisfaite au départ grâce à l'initialisation de l'étape [4] ; elle reste vraie à chaque itération (étape [3]). Ainsi lorsque la condition d'arrêt (étape [2]) est atteinte cette situation nous permet d'affirmer que le problème est résolu. C'est également en analysant l'étape [3] qu'on peut prouver la terminaison de l'algorithme.

Chapitre 3

Procédures et fonctions





Sommaire

3.1 Introduction

3.1.1 Réutiliser

Exemple 3.1 : Numération en base b

Un entier positif en base b est représenté par une suite de chiffres $(r_n r_{n-1} \ldots r_1 r_0)_b$ où les r_i sont des chiffres de la base b ($0 \leq r_i < b$). Ce nombre a pour valeur :

$$r_n b^n + r_{n-1} b^{n-1} + \ldots + r_1 b^1 + r_0 b^0 = \sum_{i=0}^{i=n} r_i b^i$$

Exemples :

$$\begin{array}{lllll} (123)_{10} & = & 1.10^2 + 2.10^1 + 3.10^0 & = & (123)_{10} \\ (123)_5 & = & 1.5^2 + 2.5^1 + 3.5^0 & = & (38)_{10} \\ (123)_8 & = & 1.8^2 + 2.8^1 + 3.8^0 & = & (83)_{10} \\ (123)_{16} & = & 1.16^2 + 2.16^1 + 3.16^0 & = & (291)_{10} \end{array}$$

On suppose que le nombre n est représenté par un tableau de chiffres (`code`) en base b ; par exemple si `b = 5` et `code = [1,2,3]`, alors en base 10 le nombre entier n correspondant vaudra 38 ($1 \cdot 5^2 + 2 \cdot 5^1 + 3 \cdot 5^0 = 25 + 10 + 3 = 38$). Etant donnés `code` et `b`, l'algorithme suivant permet de calculer n en base 10 :

```
n = 0
for i in range(len(code)):
    n = n + code[i]*b**(len(code)-1-i)
```

On ajoute successivement les puissances de la base $r_i \cdot b^i$ (`code[i]*b**(len(code)-1-i)`).

■ TD3.1

Sous l'interpréteur Python, pour calculer successivement la valeur décimale n des nombres $(123)_5$ et $(123)_8$, nous devrons donc recopier 2 fois l'algorithme ci-dessus.

```
>>> b = 5
>>> code = [1,2,3]
>>> n = 0
>>> for i in range(len(code)):
...     n = n + code[i]*b**(len(code)-1-i)
...
>>> n
38
```

```
>>> b = 8
>>> code = [1,2,3]
>>> n = 0
>>> for i in range(len(code)):
...     n = n + code[i]*b**(len(code)-1-i)
...
>>> n
83
```

Dans la pratique, on ne souhaite pas recopier 2 fois le même code d'autant plus si celui-ci nécessite de très nombreuses lignes de code. Pour améliorer la réutilisabilité de l'algorithme (figure 3.1) autrement que par un « copier-coller », une solution sera l'encapsulation du code à répéter au

Remarque 3.1 : *Les chiffres r_i en base b sont tels que $0 \leq r_i < b$.*
Exemples :

$b = 2$: $r_i \in \{0, 1\}$ (bit = **bi**nary digi**t**)

$b = 10$: $r_i \in \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$

$b = 16$: $r_i \in \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f\}$

Plus la base b est faible plus il faut de chiffres pour représenter un même nombre.
Exemples :

$$\begin{array}{lllll} (128)_{10} & = & 1.10^2 + 2.10^1 + 8.10^0 & = & (128)_{10} \\ & = & 1.2^7 & = & (10000000)_2 \\ & = & 8.16^1 & = & (80)_{16} \end{array}$$

TD 3.1 : Codage des entiers positifs (1)
Définir un algorithme qui code sur k chiffres en base b un entier positif n du système décimal.
Exemples : $(38)_{10} \rightarrow (123)_5$
$(83)_{10} \rightarrow (123)_8$
$(291)_{10} \rightarrow (123)_{16}$

Fig. 3.1 : Réutilisabilité d'un algorithme
La réutilisabilité d'un algorithme est son aptitude à être réutilisé pour résoudre des tâches équivalentes à celle pour laquelle il a été conçu.

sein d'une « fonction » (figure 3.2) comme on en a l'habitude avec les fonctions mathématiques classiques : on « appelle » la fonction sinus ($\sin(x)$) plutôt que la redéfinir à chaque fois ! Ainsi, il arrivera souvent qu'une même séquence d'instructions doive être utilisée à plusieurs reprises dans un algorithme, et on souhaitera bien évidemment ne pas avoir à la reproduire systématiquement. Pour cela, le concepteur définira et utilisera ses propres « fonctions » qui viendront compléter le jeu d'instructions initial.

Fig. 3.2 : Encapsulation
L'encapsulation est l'action de mettre une chose dans une autre : on peut considérer que cette chose est mise dans une « capsule » comme on conditionne un médicament dans une enveloppe soluble (la capsule). Ici, il s'agit d'encapsuler des instructions dans une fonction ou une procédure.



3.1.2 Structurer

Exemple 3.2 : Nombres fractionnaires
*Un nombre fractionnaire (*nombre avec des chiffres après la virgule *: $(r_n r_{n-1} \ldots r_1 r_0 . r_{-1} r_{-2} \ldots)_b$) est défini sur un sous-ensemble borné, incomplet et fini des rationnels. Un tel nombre a pour valeur :*

$$r_n b^n + r_{n-1} b^{n-1} + \ldots + r_1 b^1 + r_0 b^0 + r_{-1} b^{-1} + r_{-2} b^{-2} + \ldots$$

En pratique, le nombre de chiffres après la virgule *est limité par la taille physique en machine.*

$$(r_n r_{n-1} \ldots r_1 r_0 . r_{-1} r_{-2} \ldots r_{-k})_b = \sum_{i=-k}^{i=n} r_i b^i$$

Un nombre x pourra être représenté en base b par un triplet $[s, m, p]$ tel que $x = (-1)^s \cdot m \cdot b^p$ où s représente le signe de x, m sa mantisse et p son exposant (p comme puissance) où :
- *signe s : $s = 1$ si $x < 0$ et $s = 0$ si $x \geq 0$*
- *mantisse m : $m \in [1, b[$ si $x \neq 0$ et $m = 0$ si $x = 0$*
- *exposant p : $p \in [min, max]$*

Ainsi, le codage de $x = -9.75$ en base $b = 2$ s'effectuera en 4 étapes :

1. coder le signe de x : $x = -9.75 < 0 \Rightarrow s = 1$
2. coder la partie entière de $|x|$: $9 = (1001)_2$
3. coder la partie fractionnaire de $|x|$: $0.75 = (0.11)_2$
4. et coder $|x|$ en notation scientifique normalisée : $m \in [1, 2[$
   $(1001)_2 + (0.11)_2 = (1001.11)_2 = (1.00111)_2 \cdot 2^3$
   $= (1.00111)_2 \cdot 2^{(11)_2}$

■ TD3.2

TD 3.2 : Codage d'un nombre fractionnaire
Déterminer le signe, la mantisse et l'exposant binaires du nombre fractionnaire $x = 140.8125$ en suivant les 4 étapes décrites ci-contre.

Fig. 3.3 : MODULE `math` DE PYTHON

Constante	Valeur
`pi`	*3.1415926535897931*
`e`	*2.7182818284590451*

Fonction	Résultat
`acos(x)`	*arc cosinus (en radians).*
`asin(x)`	*arc sinus (en radians).*
`atan(x)`	*arc tangente (en radians).*
`atan2(y,x)`	*arc tangente (en radians) de* `y/x`.
`ceil(x)`	*le plus petit entier* `>=` `x`.
`cos(x)`	*cosinus (en radians).*
`cosh(x)`	*cosinus hyperbolique.*
`degrees(x)`	*conversion radians → degrés.*
`exp(x)`	*exponentielle.*
`fabs(x)`	*valeur absolue.*
`floor(x)`	*le plus grand entier* `<=` `x`.
`frexp(x)`	*mantisse* `m` *and exposant* `p` *de* `x` `x = m * 2.**p`.
`hypot(x,y)`	*distance euclidienne* `sqrt(x*x + y*y)`.
`ldexp(x,i)`	`x * (2**i)`.
`log(x[,b=e])`	*logarithme en base* `b`.
`log10(x)`	*logarithme en base 10.*
`modf(x)`	*parties fractionnaire et entière de* `x`.
`pow(x,y)`	`x` *puissance* `y` *(*`x**y`*).*
`radians(x)`	*conversion degrés → radians.*
`sin(x)`	*sinus (en radians).*
`sinh(x)`	*sinus hyperbolique.*
`sqrt(x)`	*racine carrée.*
`tan(x)`	*tangente (en radians).*
`tanh(x)`	*tangente hyperbolique.*

Cette démarche conduit au résultat $x = (-1)^1 \cdot (1.00111)_2 \cdot 2^{(11)_2}$ où $s = (1)_2$, $m = (1.00111)_2$ et $p = (11)_2$.

L'approche efficace d'un problème complexe (*coder un nombre fractionnaire*) consiste souvent à le décomposer en plusieurs sous-problèmes plus simples (*coder le signe*, *coder la mantisse*, *coder l'exposant*) qui seront étudiés séparément. Ces sous-problèmes peuvent éventuellement être eux-mêmes décomposés à leur tour (*coder la partie entière*, *coder la partie fractionnaire*, *normaliser*), et ainsi de suite. Là encore, le concepteur définira et utilisera ses propres « fonctions » pour réaliser la structuration d'un problème en sous-problèmes : il divise le problème en sous-problèmes pour mieux le contrôler (*diviser pour régner*).

3.1.3 Encapsuler

Exemple 3.3 : CALCUL DE **sin(π/2)**

```
>>> from math import sin, pi
>>> sin(pi/2)
1.0
>>> y = sin(pi/2)
>>> y
1.0
```

Cette petite session PYTHON illustre quelques caractéristiques importantes des fonctions.

- Une fonction à un nom : `sin`.
- Une fonction est en général « rangée » dans une bibliothèque de fonctions (ici la bibliothèque `math` de PYTHON, figure 3.3) ; il faut aller la chercher (on « importe » la fonction) : `from math import sin`.
- Une fonction s'utilise (s'« appelle ») sous la forme d'un nom suivi de parenthèses ; dans les parenthèses, on « transmet » à la fonction un ou plusieurs arguments : `sin(pi/2)`.
- L'évaluation de la fonction fournit une valeur de retour ; on dit aussi que la fonction « renvoie » ou « retourne » une valeur (`sin(pi/2)` → `1.0`) qui peut ensuite être affectée à une variable : `y = sin(pi/2)`.

Définition 3.1 : FONCTION

Une fonction est un bloc d'instructions nommé et paramétré, réalisant une certaine tâche. Elle admet zéro, un ou plusieurs paramètres et renvoie toujours un résultat.

Une fonction en informatique se distingue principalement de la fonction mathématique par le fait qu'en plus de calculer un résultat à partir de paramètres, la fonction informatique peut

avoir des « effets de bord » : par exemple afficher un message à l'écran, jouer un son, ou bien piloter une imprimante. Une fonction qui n'a pas d'effet de bord joue le rôle d'une expression évaluable. Une fonction qui n'a que des effets de bord est appelée une procédure et joue le rôle d'une instruction.

Définition 3.2 : procédure
Une procédure est un bloc d'instructions nommé et paramétré, réalisant une certaine tâche. Elle admet zéro, un ou plusieurs paramètres et ne renvoie pas de résultat.

Les procédures et les fonctions intégrées au langage sont relativement peu nombreuses : ce sont seulement celles qui sont susceptibles d'être utilisées très fréquemment (l'annexe 3.5.2 page 153 présente les principales fonctions intégrées systématiquement avec l'interpréteur Python). Les autres fonctions sont regroupées dans des fichiers séparés que l'on appelle des modules. Les modules sont donc des fichiers qui regroupent des ensembles de fonctions. Souvent on regroupe dans un même module des ensembles de fonctions apparentées que l'on appelle des bibliothèques. Pour pouvoir utiliser ces fonctions, il faut importer le module correspondant.

Ainsi en Python, il existe de nombreux modules additionnels dont le plus connu est le module `math` qui définit une vingtaine de constantes et fonctions mathématiques usuelles (figure 3.3). On peut importer ce module de différentes manières :

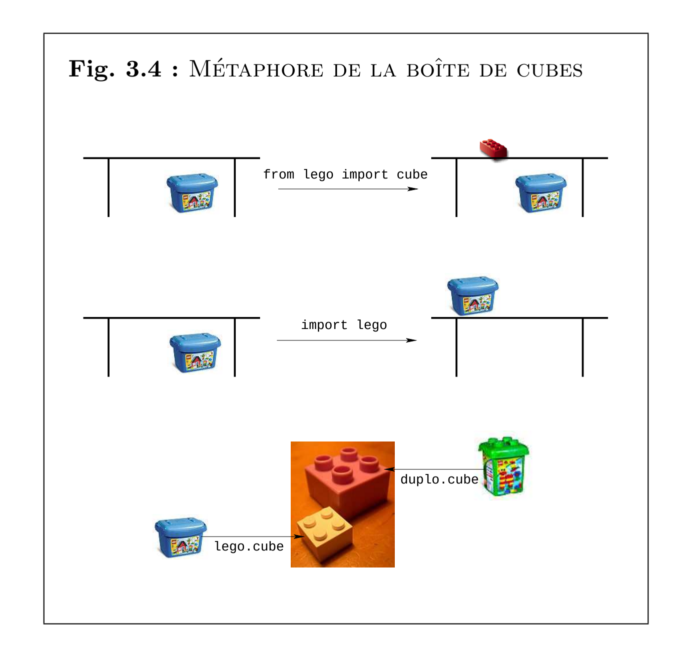
```
>>> from math import sin, pi
>>> sin(pi/2)
1.0
```

```
>>> import math
>>> math.sin(math.pi/2)
1.0
```

Pour comprendre la différence entre ces deux méthodes, aidons-nous de la métaphore des cubes *à la* Lego (figure 3.4). Nous disposons d'une boîte de cubes (`lego`) rangée sous la table. Pour utiliser un cube de cette boîte, nous pouvons soit prendre le cube dans la boîte et le mettre sur la table (`from lego import cube`) : le cube est alors directement accessible sur la table (`cube`) ; soit mettre la boîte sur la table (`import lego`) : le cube n'est alors pas directement accessible, il faut encore le prendre dans la boîte (`lego.cube`). L'intérêt de cette deuxième méthode est de distinguer 2 cubes qui porteraient le même nom (`cube`) mais qui ne seraient pas originaires de la même boîte (boîtes `lego` et `duplo`) et qui seraient donc différents (`lego.cube` et `duplo.cube`). Il est possible également de verser tout le contenu de la boîte sur la table (`from lego import *`) : ici, l'astérisque (`*`) signifie « tout ».

```
>>> from math import *
>>> sqrt(tan(log(pi)))
1.484345173593278
```

Fig. 3.4 : Métaphore de la boîte de cubes



Remarque 3.2 : Lego *est une société danoise fabriquant des jeux dont le produit phare est un jeu de briques en plastique à assembler.*

3.2 Définition d'une fonction

Pour encapsuler un algorithme dans une fonction, on suivra pas à pas la démarche suivante :

1. donner un nom explicite à l'algorithme,
2. définir les paramètres d'entrée-sortie de l'algorithme,
3. préciser les préconditions sur les paramètres d'entrée,
4. donner des exemples d'utilisation et les résultats attendus,
5. décrire par une phrase ce que fait l'algorithme et dans quelles conditions il le fait,
6. encapsuler l'algorithme dans la fonction spécifiée par les 5 points précédents.

Les 5 premières étapes relèvent de la spécification de l'algorithme et la dernière étape concerne l'encapsulation proprement dite de l'algorithme. En Python, la spécification sera exécutable : à chaque étape, le code de la fonction est toujours exécutable même s'il ne donne pas encore le bon résultat ; seule la dernière étape d'encapsulation (voir section 3.2.6 page 112) permettra d'obtenir le résultat valide attendu.

Définition 3.3 : SPÉCIFICATION D'UN ALGORITHME
La spécification d'un algorithme décrit ce que fait l'algorithme et dans quelles conditions il le fait.

Définition 3.4 : IMPLÉMENTATION D'UN ALGORITHME
L'implémentation d'un algorithme décrit comment fait l'algorithme pour satisfaire sa spécification.

Exemple 3.4 : NOMBRES DE FIBONACCI
La fonction de Fibonacci calcule le nombre u_n *à l'ordre* n *(dit de Fibonacci) selon la relation de récurrence :*

$$u_0 = 1 \ , \ u_1 = 1 \ , \ u_n = u_{n-1} + u_{n-2} \ \forall n \in N, \ n > 1$$

Les 10 premiers nombres de Fibonacci valent donc : $u_0 = 1$, $u_1 = 1$, $u_2 = 2$, $u_3 = 3$, $u_4 = 5$, $u_5 = 8$, $u_6 = 13$, $u_7 = 21$, $u_8 = 34$ *et* $u_9 = 55$.

Dans cette section, nous utiliserons cet exemple comme fil conducteur dans la définition d'une fonction. Ainsi donc, nous voulons définir une fonction qui calcule le $n^{ième}$ nombre de Fibonacci. Cette description de la fonction (« calculer le $n^{ième}$ nombre de Fibonacci ») évoluera progressivement à chaque étape et deviendra suffisamment précise pour qu'un autre utilisateur puisse l'utiliser effectivement sans surprise et en toute sécurité.

3.2.1 Nommer

La première chose à faire est de nommer la fonction qui encapsule l'algorithme. Les noms de fonction sont des identificateurs arbitraires, de préférence assez courts mais aussi explicites que possible, de manière à exprimer clairement ce que la fonction est censée faire. Les noms des fonctions doivent en outre obéir à quelques règles simples :

– Un nom de fonction est une séquence de lettres (a...z , A...Z) et de chiffres (0...9), qui doit toujours commencer par une lettre.
– Seules les lettres ordinaires sont autorisées. Les lettres accentuées, les cédilles, les espaces, les caractères spéciaux tels que `$`, `#`, `@`, etc. sont interdits, à l'exception du caractère _ (souligné).
– La « casse » est significative : les caractères majuscules et minuscules sont distingués. Ainsi, `sinus`, `Sinus`, `SINUS` sont des fonctions différentes.
– Par convention, on écrira l'essentiel des noms de fonction en caractères minuscules (y compris la première lettre). On n'utilisera les majuscules qu'à l'intérieur même du nom pour en augmenter éventuellement la lisibilité, comme dans `triRapide` ou `triFusion`.
– Le langage lui-même peut se réserver quelques noms comme c'est le cas pour PYTHON (figure 3.5). Ces mots réservés ne peuvent donc pas être utilisés comme noms de fonction.

En ce qui concerne la fonction de Fibonacci de l'exemple 3.4, nous choisissons de l'appeler simplement `fibonacci`. Ce qui se traduira en PYTHON par l'en-tête suivante où `def` et `return` sont deux mots réservés par PYTHON pour définir les fonctions (voir annexe 3.5.3 page 155) :

```
def fibonacci():
    return
```

```
>>> from fibo import fibonacci
>>> fibonacci()
>>>
```

Le code de la partie gauche est la définition actuelle de la fonction `fibonacci` que l'on a éditée dans un fichier `fibo.py` : le module associé a donc pour nom `fibo`. La partie droite montre comment on utilise couramment le module `fibo` et la fonction `fibonacci` sous l'interpréteur PYTHON. Dans l'état actuel, cette fonction n'a pas d'arguments et ne fait rien ! Mais elle est déjà compilable et exécutable. On peut la décrire de la manière suivante : « La fonction `fibonacci` calcule un nombre de Fibonacci ».

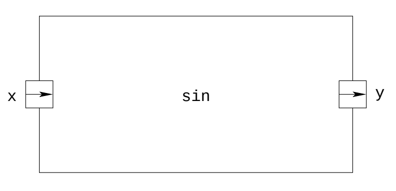
Fig. 3.5 : MOTS RÉSERVÉS EN PYTHON

```
and       del      for     is      raise
assert    elif     from    lambda  return
break     else     global  not     try
class     except   if      or      while
continue  exec     import  pass    with
def       finally  in      print   yield
```

3.2.2 Paramétrer

Un algorithme est une suite ordonnée d'instructions qui indique la démarche à suivre pour résoudre une série de problèmes équivalents. Dans ce contexte, c'est le paramétrage de l'algorithme qui lui donnera cette capacité recherchée de résoudre des problèmes équivalents. Dans l'exemple de la fonction `sin`, c'est en effet le paramètre `x` qui permet de calculer le sinus de n'importe quel nombre réel (`sin(x)`) et non le sinus d'un seul nombre. La deuxième étape de la définition d'une fonction consiste donc à préciser les paramètres d'entrée-sortie de la fonction (figure 3.6).

Fig. 3.6 : Paramètres d'une fonction



`sin` *est le nom de la fonction ;* `x` *est le seul paramètre d'entrée,* `y` *le seul paramètre de sortie.*

`y = sin(x)`

a
b
c
functionName
s
t

`functionName` *est le nom de la fonction ;* `a`*,* `b` *et* `c` *sont les 3 paramètres d'entrée,* `s` *et* `t` *les 2 paramètres de sortie.*

`(s,t) = functionName(a,b,c)`

Définition 3.5 : paramètre d'entrée
Les paramètres d'entrée d'une fonction sont les arguments de la fonction qui sont nécessaires pour effectuer le traitement associé à la fonction.

Définition 3.6 : paramètre de sortie
Les paramètres de sortie d'une fonction sont les résultats retournés par la fonction après avoir effectué le traitement associé à la fonction.

Pour cela, on nomme ces paramètres avec des identificateurs explicites dans le contexte courant d'utilisation de la fonction. Les paramètres de sortie seront systématiquement initialisés à une valeur par défaut (par exemple : `0` pour un entier, `False` pour un booléen, `0.0` pour un réel, `''` pour une chaîne de caractères, `[]` pour un tableau).

La fonction `fibonacci` prend l'ordre `n` en paramètre d'entrée et retourne le nombre `u` (u_n). Le paramètre de sortie `u` est un entier qui sera donc *a priori* initialisé à `0` dans la fonction et qui sera retourné par la fonction (`return u`).

```
def fibonacci(n):
    u = 0
    return u
```

```
>>> from fibo import fibonacci
>>> fibonacci(5)
0
>>> fibonacci(-5)
0
>>> fibonacci('n')
0
```

Le code de la partie gauche est la nouvelle définition de la fonction `fibonacci` toujours éditée dans le même fichier `fibo.py` que précédemment. La partie droite montre son utilisation sous l'interpréteur Python. Dans l'état actuel, cette fonction retourne `0` quels que soient le type et la valeur du paramètre d'entrée ! Mais elle est encore compilable et exécutable. On peut

maintenant la décrire de la manière suivante : « `u = fibonacci(n)` est le $n^{ième}$ nombre de Fibonacci » (figure 3.7). Cette description est un peu moins « littéraire » que dans la section 3.2.1 précédente. Elle a cependant l'avantage de montrer l'utilisation typique de la fonction (`u = fibonacci(n)`) et d'expliciter le sens des paramètres d'entrée-sortie (`n` et `u`).

3.2.3 Protéger

Dans la section précédente, nous avons testé la fonction `fibonacci` avec comme paramètre d'entrée la chaîne de caractères `'n'` : cela n'a aucun sens pour le calcul d'un nombre de Fibonacci ! Il faut donc protéger la fonction pour la rendre plus robuste face à des contextes anormaux d'utilisation (figure 3.8). Pour cela, on imposera aux paramètres d'entrée de vérifier certaines conditions avant d'exécuter la fonction appelée. Ces conditions préalables à l'exécution sont appelées les préconditions.

Définition 3.7 : PRÉCONDITION
Les préconditions d'une fonction sont les conditions que doivent impérativement vérifier les paramètres d'entrée de la fonction juste avant son exécution.

Une précondition est donc une expression booléenne qui prend soit la valeur `False` (faux) soit la valeur `True` (vrai). Elle peut contenir des opérateurs de comparaison (`==`, `!=`, `>`, `>=`, `<=`, `<`), des opérateurs logiques (`not`, `and`, `or`), des opérateurs d'identité (`is`, `is not`), des opérateurs d'appartenance (`in`, `not in`) ou toutes autres fonctions booléennes. En PYTHON, on définira les préconditions que doivent vérifier les paramètres d'entrée de la fonction à l'aide de la directive `assert` (figure 3.9). A l'exécution du code, cette directive « lèvera une exception » si la condition (l'assertion, figure 3.10) testée est fausse. Les préconditions seront placées systématiquement juste après l'en-tête de la fonction (`def fibonacci(n) :`).

```
def fibonacci(n):
    assert type(n) is int
    assert n >= 0
    u = 0
    return u
```

```
>>> fibonacci(-5)
Traceback ...
    assert n >= 0
AssertionError
>>> fibonacci('n')
Traceback ...
    assert type(n) is int
AssertionError
```

La définition de la fonction `fibonacci` a donc été complétée par les préconditions sur le paramètre d'entrée : n doit être un entier (`type(n) is int`) positif ou nul (`n >= 0`). Sa descrip-

Fig. 3.7 : FONCTION `fibonacci` (1)

`fibonacci` *est le nom de la fonction ;* `n` *est le seul paramètre d'entrée,* `u` *le seul paramètre de sortie.*

`u = fibonacci(n)`

Fig. 3.8 : ROBUSTESSE D'UN ALGORITHME
La robustesse d'un algorithme est son aptitude à se protéger de conditions anormales d'utilisation.

Fig. 3.9 : L'INSTRUCTION `assert` EN PYTHON
`assert expr[, message]`

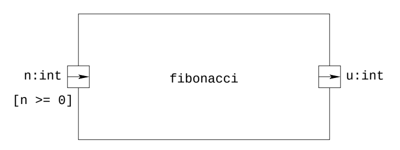
`expr` *est évaluée : si* `expr == True`*, on passe à l'instruction suivante, sinon l'exécution est interrompue et une exception* `AssertionError` *est levée qui affiche le* `message` *optionnel.*
Exemple :

```
>>> assert False, "message d'erreur"
Traceback (most recent call last) :
  File "<stdin>", line 1, in <module>
AssertionError : message d'erreur
>>>
```

tion peut alors être complétée pour tenir compte de ces préconditions sur l'ordre `n` du nombre de Fibonacci calculé : « `u = fibonacci(n)` est le $n^{ième}$ nombre de Fibonacci si `n :int >= 0` » (figure 3.11). La fonction est toujours compilable et exécutable, mais son exécution est maintenant systématiquement interrompue si les préconditions ne sont pas respectées : ce qui est le cas pour les paramètres d'entrée `-5` et `'n'`. Dans tous les autres cas (entiers positifs ou nuls), elle retourne toujours `0` !

Fig. 3.10 : Définition de l'Académie (8)
ASSERTION *n. f. XIIIe siècle. Emprunté du latin assertio, dérivé de assertum, supin de asserere, « revendiquer », puis « prétendre, affirmer ». Proposition qu'on avance et qu'on soutient comme vraie.*

Fig. 3.11 : Fonction `fibonacci` (2)



`n :int` *est le paramètre d'entrée de nom* `n` *et de type* `int` *qui doit vérifier la précondition* `[n >= 0]`.
`u :int` *est la paramètre de sortie de nom* `u` *et de type* `int`.

`u = fibonacci(n)`

En Python, la manière dont nous avons initialisé le paramètre de sortie `u` (`u = 0`) indique qu'il s'agit implicitement d'un entier (`int`). La fonction `fibonacci` retourne donc un entier : c'est une postcondition sur le paramètre de sortie `u` dans le cas du calcul d'un nombre de Fibonacci.

Définition 3.8 : postcondition
Les postconditions d'une fonction sont les conditions que doivent impérativement vérifier les paramètres de sortie de la fonction juste après son exécution.

En plus des préconditions et des postconditions, on pourra quelquefois imposer que des conditions soient vérifiées tout au long de l'exécution de la fonction : on parle alors d'invariants.

Définition 3.9 : invariant
Les invariants d'une fonction sont les conditions que doit impérativement vérifier la fonction tout au long de son exécution.

De tels exemples d'invariants seront mis en évidence lors de l'implémentation de certaines fonctions.

3.2.4 Tester

Avant même d'implémenter la fonction proprement dite (voir section 3.2.6), on définit des tests que devra vérifier la fonction une fois implémentée. Ces tests sont appelés tests unitaires car ils ne concernent qu'une seule fonction, la fonction que l'on cherche à définir. Ce jeu de tests constitue ainsi un ensemble caractéristique d'entrées-sorties associées que devra vérifier la fonction. Par exemple, `fibonacci(0)` devra retourner `1`, `fibonacci(1)` devra retourner `1`, `fibonacci(2)` devra retourner `2` ou encore `fibonacci(9)` devra retourner `55`. En fait, quelle que soit la manière dont sera implémentée la fonction `fibonacci`, les résultats précédents devront être obtenus par cette implémentation.

En Python, on utilisera une `docstring` (chaîne entre 3 guillemets : `""" ... """`) pour décrire ces tests. Cette chaîne spéciale, placée entre l'en-tête et les préconditions et qui peut

tenir sur plusieurs lignes, joue le rôle de commentaire dans le corps de la fonction. Elle ne change donc *a priori* rien à son exécution courante.

```
def fibonacci(n):
    """
    >>> fibonacci(0)
    1
    >>> fibonacci(2)
    2
    >>> fibonacci(9)
    55
    """
    assert type(n) is int
    assert n >= 0
    u = 0
    return u
```

```
>>> fibonacci(0)
0
>>> fibonacci(2)
0
>>> fibonacci(9)
0
```

Nous avons donc ajouter 3 tests dans la définition de la fonction `fibonacci`. En PYTHON, ces tests ont la particularité de se présentent sous la même forme que lorsqu'on appelle la fonction sous l'interpréteur PYTHON, à ceci près qu'ils sont écrits dans une `docstring` (`""" ... """`) :

```
"""
>>> fibonacci(0)
1
>>> fibonacci(2)
2
>>> fibonacci(9)
55
"""
```

```
>>> fibonacci(0)
0
>>> fibonacci(2)
0
>>> fibonacci(9)
0
```

La fonction est toujours compilable et exécutable, et son exécution retourne toujours `0`. Si maintenant, nous ajoutons les 3 lignes ci-dessous à la fin du fichier `fibo.py`, les tests que nous avons ajoutés à la définition de la fonction `fibonacci` vont être évalués automatiquement (figure 3.12) comme le montre l'exécution de la page suivante.

```
if __name__ == "__main__" :
    import doctest
    doctest.testmod()
```

D'une certaine manière ces tests permettent de préciser ce qu'on attend de la fonction. Le choix de ces tests est donc très important pour valider l'implémentation future.

Fig. 3.12 : LE MODULE `doctest`
Le module `doctest` *est un module standard* PYTHON *qui offre des services pour manipuler les* `docstring`*s utilisées dans les jeux de tests :*
www.python.org/doc/2.5/lib/module-doctest.html

The doctest module searches for pieces of text that look like interactive PYTHON *sessions, and then executes those sessions to verify that they work exactly as shown. There are several common ways to use doctest :*
- *To check that a module's docstrings are up-to-date by verifying that all interactive examples still work as documented.*
- *To perform regression testing by verifying that interactive examples from a test file or a test object work as expected.*
- *To write tutorial documentation for a package, liberally illustrated with input-output examples. Depending on whether the examples or the expository text are emphasized, this has the flavor of "literate testing" or "executable documentation".*

The functions `testmod()` *and* `testfile()` *provide a simple interface to* `doctest` *that should be sufficient for most basic uses.*

```
$ python fibo.py
**********************************************
File "fibo.py", line 10, in __main__.fibonacci
Failed example:
    fibonacci(0)
Expected:
    1
Got:
    0
**********************************************
File "fibo.py", line 12, in __main__.fibonacci
Failed example:
    fibonacci(2)
Expected:
    2
Got:
    0
**********************************************
File "fibo.py", line 14, in __main__.fibonacci
Failed example:
    fibonacci(9)
Expected:
    55
Got:
    0
**********************************************
1 items had failures:
   3 of   3 in __main__.fibonacci
***Test Failed*** 3 failures.
$
```

On lance l'interpréteur PYTHON en passant le fichier à tester (ici `fibo.py`). Chaque test du jeu de tests placé dans la `docstring` de la fonction à tester sera exécuté par l'interpréteur. Si le résultat du test est conforme à ce qui était prévu (par exemple `factorielle(0)` → `1`), rien de particulier ne se passe. Si le test est en erreur, PYTHON précise la ligne du fichier où apparaît le test (par exemple : `File "fibo.py", line 10`), le test en erreur (`fibonacci(0)`), le résultat attendu (`Expected : 1`) et le résultat obtenu (`Got : 0`). Ainsi, on peut vérifier simplement si la fonction a passé le jeu de tests. C'est une manière de tester la validité de l'implémentation.

Ces tests seront conservés tant que la fonctionnalité est requise. A chaque modification du code, on effectuera tous les tests ainsi définis pour vérifier si quelque chose ne fonctionne plus.

3.2.5 Décrire

Une fois choisis le nom de la fonction, les paramètres d'entrée-sortie, les préconditions sur les paramètres d'entrée et les jeux de tests, on peut alors préciser « ce que fait l'algorithme » et « dans quelles conditions il le fait » : il s'agit d'une phrase de description « en bon français » qui permettra à tout utilisateur de comprendre ce que fait l'algorithme sans nécessairement savoir

comment il le fait.

Définition 3.10 : DESCRIPTION D'UNE FONCTION
La description d'une fonction est une phrase qui précise ce que fait la fonction et dans quelles conditions elle le fait.

Cette phrase est une chaîne de caractères qui doit expliciter le rôle des paramètres d'entrée et leurs préconditions, ainsi que toutes autres informations jugées nécessaires par le concepteur de la fonction. En particulier, lorsque le retour de la fonction n'est pas « évident », on explicitera les paramètres de sortie. Dans le cas de la fonction `fibonacci`, un utilisateur de la fonction « saura » ce qu'est un nombre de Fibonacci et la précision que cette fonction retourne un entier n'est pas nécessaire. Par contre, si la fonction retourne plus d'une valeur, il faut au moins préciser l'ordre dans lequel elle les retourne. Ainsi par exemple, pour la fonction `divmod(a,b)` de la bibliothèque standard PYTHON qui calcule le quotient `q` et le reste `r` de la division entière $a \div b$ (voir annexe 3.5.2 page 153), il faut préciser qu'elle retourne un n-uplet (*tuple*) dans l'ordre `(q,r)`. Dans certains cas, il faut également préciser que la fonction effectue les calculs dans telles ou telles unités : c'est par exemple le cas des fonctions trigonométriques usuelles où les calculs sont menés avec des angles en radians (figure 3.3 page 102). Dans d'autres cas encore, on pourra préciser une référence bibliographique ou un site WEB où l'utilisateur pourra trouver des compléments sur la fonction et l'algorithme associé. La description d'une fonction intègrera donc au moins :

- un exemple typique d'appel de la fonction,
- la signification des paramètres d'entrée-sortie,
- les préconditions sur les paramètres d'entrée,
- un jeu de tests significatifs.

Ainsi, la description de la fonction `fibonacci` pourra se présenter sous la forme suivante :
« `u = fibonacci(n)` est le nombre de Fibonacci à l'ordre `n` si `n :int >= 0`.
Exemples : `fibonacci(0)` → `1`, `fibonacci(2)` → `2`, `fibonacci(9)` → `55` ».

En PYTHON, on intègrera cette description dans la `docstring` (chaîne entre 3 guillemets) que l'on a déjà utilisée pour le jeu de tests. Les exemples seront donnés sous la forme d'un jeu de tests *à la* PYTHON. Cette chaîne spéciale, qui peut tenir sur plusieurs lignes, joue le rôle de commentaire dans le corps de la fonction. Elle ne change donc rien à son exécution. Par contre, elle permettra de documenter automatiquement l'aide en-ligne de PYTHON (`help`) ou encore, elle pourra être utilisée par certains environnements (`idle` par exemple) ou par certains utilitaires comme `pydoc` (voir annexe 3.5.4 page 156).

```
def fibonacci(n):
    """
    u = fibonacci(n)
    est le nombre de Fibonacci
    à l'ordre n si n:int >= 0
    >>> fibonacci(0)
    1
    >>> fibonacci(2)
    2
    >>> fibonacci(9)
    55
    """
    assert type(n) is int
    assert n >= 0
    u = 0
    return u
```

```
>>> fibonacci(5)
0
>>> fibonacci(100)
0
>>> help(fibo.fibonacci)
Help on function fibonacci in module fibo:

fibonacci(n)
    u = fibonacci(n)
    est le nombre de Fibonacci
    à l'ordre n si n:int >= 0
    >>> fibonacci(0)
    1
    >>> fibonacci(2)
    2
    >>> fibonacci(9)
    55
```

La fonction est toujours compilable et exécutable ; elle est maintenant documentable (`help(fibo.fibonacci)`), mais retourne toujours `0` ! Il reste à l'implémenter. ■ TD3.3

3.2.6 Encapsuler

La dernière étape consiste enfin à dire « comment fait la fonction » pour répondre à la spécification décrite au cours des étapes précédentes. En phase de spécification, la fonction (ou la procédure) est vue comme une boîte noire dont on ne connaît pas le fonctionnement interne (figure 3.13 : spécification). En phase d'implémentation, l'enchaînement des instructions nécessaires à la résolution du problème considéré est détaillé (figure 3.13 : implémentation).

```
>>> n
9
>>> u, u1, u2 = 1, 1, 1
>>> for i in range(2,n+1):
...     u = u1 + u2
...     u2 = u1
...     u1 = u
...
>>> u
55
```

Compte-tenu de la définition des nombres de Fibonacci (exemple 3.4), les instructions ci-contre permettent de calculer le nombre `u` de Fibonacci pour un ordre `n` donné (ici `9`). Pour chaque valeur de `i`, `u1` représente initialement le nombre de Fibonacci à l'ordre `(i-1)`, `u2` le nombre de Fibonacci à l'ordre `(i-2)` et `u` (`= u1 + u2`) le nombre de Fibonacci à l'ordre `i`.

Ce sont ces instructions qu'il faut encapsuler au cœur de la fonction `fibonacci` comme le

TD 3.3 : Décodage base b → décimal

En section 3.1.1, la valeur décimale d'un nombre entier codé en base b était obtenu par l'algorithme suivant :

```
>>> n = 0
>>> for i in range(len(code)) :
...     n = n + code[i]*b**(len(code)-1-i)
...
>>>
```

Spécifier une fonction qui encapsulera cet algorithme.

Fig. 3.13 : Fonction `fibonacci` (3)

Spécification :

`u = fibonacci(n)` *est le nombre de Fibonacci à l'ordre* `n` *si* `n :int >= 0`.

Exemples : `fibonacci(0)` → `1`
`fibonacci(2)` → `2`
`fibonacci(9)` → `55`

Implémentation :

montre le code complet ci-dessous (voir remarque 3.3). ■TD3.4

```
def fibonacci(n):
    """
    u = fibonacci(n)
    est le nombre de Fibonacci
    à l'ordre n si n:int >= 0
    >>> fibonacci(0)
    1
    >>> fibonacci(2)
    2
    >>> fibonacci(9)
    55
    """
    assert type(n) is int
    assert n >= 0
    u, u1, u2 = 1, 1, 1
    for i in range(2,n+1):
        u = u1 + u2
        u2 = u1
        u1 = u
    return u
```

```
>>> from fibo import fibonacci
>>> for n in range(10):
...     print(fibonacci(n),end=' ')
...
1 1 2 3 5 8 13 21 34 55
>>> for n in range(10,60,10):
...     print(fibonacci(n),end=' ')
...
89 10946 1346269 165580141 20365011074
>>> fibonacci(100)
573147844013817084101L
>>> fibo.fibonacci(120)
8670007398507948658051921L
>>> fibonacci(200)
453973694165307953197296969697410619233826L
```

En fait, plusieurs implémentations peuvent correspondre à la même spécification. C'est le cas par exemple pour le calcul de la somme des n premiers nombres entiers ci-dessous.

```
def sommeArithmetique(n):
    """
    s = sommeArithmetique(n) est la somme
    des n premiers entiers si n:int >= 0
    >>> for n in range(7):
    ...     print(sommeArithmetique(n)\
                  == n*(n+1)/2,end=' ')
    True True True True True True True
    """
    assert type(n) is int
    assert n >= 0

    s = n*(n+1)/2

    return s
```

```
def sommeArithmetique(n):
    """
    s = sommeArithmetique(n) est la somme
    des n premiers entiers si n:int >= 0
    >>> for n in range(7):
    ...     print(sommeArithmetique(n)\
                  == n*(n+1)/2,end=' ')
    True True True True True True True
    """
    assert type(n) is int
    assert n >= 0

    s = 0
    for k  in range(1,n+1): s = s + k

    return s
```

L'implémentation de gauche repose directement sur le résultat mathématique bien connu

Remarque 3.3 : *La suite de Fibonacci doit son nom au mathématicien italien Fibonacci (1175-1250). Dans un problème récréatif, Fibonacci décrit la croissance d'une population « idéale » de lapins de la manière suivante :*

- *le premier mois, il y a juste une paire de lapereaux ;*
- *les lapereaux ne sont pubères qu'à partir du deuxième mois ;*
- *chaque mois, tout couple susceptible de procréer engendre effectivement un nouveau couple de lapereaux ;*
- *les lapins ne meurent jamais !*

Se pose alors le problème suivant :

> « Possédant initialement un couple de lapins, combien de couples obtient-on en douze mois si chaque couple engendre tous les mois un nouveau couple à compter du second mois de son existence ? »

Ce problème est à l'origine de la suite de Fibonacci dont le $n^{ième}$ terme correspond au nombre de couples de lapins au $n^{ième}$ mois. Au bout d'un an, il y aura donc `fibonacci(12)` *(= 233) couples de lapins dans cette population « idéale ». On n'ose à peine imaginer ce que serait cet univers « idéal » de lapins au bout de 10 ans (*`8670007398507948658051921` *couples !)...*

TD 3.4 : Codage des entiers positifs (2)
Définir (spécifier et implémenter) une fonction qui code un entier n en base b sur k chiffres (voir TD 3.1).

Remarque 3.4 : *La somme S des n premiers nombres entiers est telle que :*

$$S = \sum_{k=1}^{n} k = (1 + 2 + 3 + \cdots + n) = \frac{n(n+1)}{2}$$

En effet :

$$\begin{array}{ccccccccc} S & = & 1 & + & 2 & + & \cdots & + & n \\ S & = & n & + & n-1 & + & \cdots & + & 1 \\ \hline 2S & = & (n+1) & + & (n+1) & + & \cdots & + & (n+1) \end{array}$$

d'où : $2S = n(n+1)$

TD 3.5 : Une spécification, des implémentations

1. *Proposer deux implémentations du calcul de la somme $s = \sum_0^n u_k$ des n premiers termes d'une suite géométrique $u_k = a \cdot b^k$.*
2. *Comparer les complexités de ces deux implémentations.*

concernant les suites arithmétiques (`s = n*(n+1)/2` : voir remarque 3.4). L'implémentation de droite effectue la somme 1 à 1 des n premiers éléments (`s = s + k`). Ces deux implémentations sont toutes les deux conformes au jeu de tests, mais elles ne se valent pas en terme de complexité. La première est dite « à temps constant » : quelle que soit la valeur de n, le calcul ne nécessitera jamais plus de 3 opérations (`(n+1)`, `n*(n+1)` et `n*(n+1)/2`). La seconde est dite « à temps linéaire » : le nombre d'opérations (`s + k`) dépend linéairement de n (il y a $(n-1)$ additions à calculer pour $n > 1$). La première implémentation est donc plus efficace que la deuxième et lui sera préférée. ■TD3.5

Ce sera le rôle du concepteur d'un algorithme de définir sa spécification et d'en proposer une implémentation efficace. L'utilisateur de la fonction, quant à lui, n'a pas à connaître l'implémentation ; seule la spécification de la fonction le concerne car elle lui est nécessaire pour appeler la fonction.

3.2.7 Conclure

L'algorithmique, ou science des algorithmes, s'intéresse à l'art de construire des algorithmes ainsi qu'à caractériser leur validité, leur robustesse leur réutilisabilité, leur complexité ou encore leur efficacité. Certaines de ces caractéristiques générales (validité, robustesse, réutilisabilité) se concrètisent à la lumière des préconisations précédentes concernant la définition d'une fonction.

Validité : La validité d'un algorithme est son aptitude à réaliser exactement la tâche pour laquelle il a été conçu.

ie : L'implémentation de la fonction doit être conforme aux jeux de tests.

Robustesse : La robustesse d'un algorithme est son aptitude à se protéger de conditions anormales d'utilisation.

ie : La fonction doit vérifier impérativement ses préconditions.

Réutilisabilité : La réutilisabilité d'un algorithme est son aptitude à être réutilisé pour résoudre des tâches équivalentes à celle pour laquelle il a été conçu.

ie : La fonction doit être correctement paramétrée.

Une fois la fonction définie (spécifiée et implémentée), il reste à l'utiliser (à l'appeler).

3.3 Appel d'une fonction

Une fonction s'utilise (s'« appelle ») sous la forme du nom de la fonction suivi de parenthèses à l'intérieur desquelles on transmet (on « passe ») zéro ou plusieurs arguments conformément à la spécification de la fonction. Dans le cas de la fonction `fibonacci`, on doit transmettre un argument et un seul lors de l'appel de la fonction.

```
>>> fibonacci(9)
55
>>> fibonacci(5+4)
55
>>> fibonacci(8) + fibonacci(7)
55
```

```
>>> y = fibonacci(9)
>>> y
55
>>> x = 9
>>> y = fibonacci(x)
>>> y
55
```

L'argument transmis peut tout aussi bien être une constante (par exemple `9`), une variable (par exemple `x`) ou toute expression dont la valeur est un entier positif ou nul (par exemple `5+4`). Mais quel rapport existe-t-il entre ces arguments transmis et les paramètres d'entrée-sortie de la fonction (`n` et `u` dans le cas de la fonction `fibonacci`) ? C'est le problème du « passage des paramètres » de l'instruction appelante à la fonction appelée.

3.3.1 Passage des paramètres

Dans la définition d'une fonction, la liste des paramètres d'entrée spécifie les informations à transmettre comme arguments lors de l'appel de la fonction. Les arguments effectivement utilisés doivent être fournis dans le même ordre que celui des paramètres d'entrée correspondants : le premier argument sera affecté au premier paramètre d'entrée, le second argument sera affecté au second paramètre d'entrée, et ainsi de suite.

Définition :

```
def fibonacci(n):
    ...
    return u
```

Appel :

```
>>> y = fibonacci(x)
```

Les paramètres introduits dans la définition sont appelés les paramètres formels (par exemple `n` dans la définition de la fonction `fibonacci`) ; les paramètres passés en arguments sont appelés les paramètres effectifs (par exemple `x` dans l'appel `fibonacci(x)`).

Définition 3.11 : PARAMÈTRE FORMEL
Un paramètre formel est un paramètre d'entrée d'une fonction utilisé à l'intérieur de la fonction appelée.

Définition 3.12 : PARAMÈTRE EFFECTIF
Un paramètre effectif est un paramètre d'appel d'une fonction utilisé à l'extérieur de la fonction appelée.

Lors de l'appel d'une fonction, il y a copie des paramètres effectifs dans les paramètres formels : les valeurs des paramètres effectifs passés en argument sont affectées aux paramètres formels d'entrée correspondants (on parle de « passage par valeur » ou de « passage par copie »). Ainsi, ce ne sont pas les paramètres effectifs qui sont manipulés par la fonction elle-même mais des copies de ces paramètres. A la sortie de la fonction, on copie la valeur du paramètre formel de retour (`u` dans la fonction `fibonacci`) dans un paramètre effectif qui remplace temporairement l'appel de la fonction (`fibonacci(x)`) et qui est finalement utilisé par l'instruction qui a appelé la fonction (l'affectation dans le cas de l'instruction `y = fibonacci(x)`). Cette variable temporaire est ensuite détruite automatiquement. Ainsi, l'instruction `y = fibonacci(x)` est « équivalente » à la séquence d'instructions suivante : ■TD3.6

Définition :
```
def fibonacci(n) :
    u, u1, u2 = 1, 1, 1
    for i in range(2,n+1) :
        u = u1 + u2
        u2 = u1
        u1 = u
    return u
```

Appel :
```
>>> x = 9
>>> y = fibonacci(x)
>>> y
55
```

Appel équivalent :
```
>>> x = 9
>>> n = x
>>> u, u1, u2 = 1, 1, 1
>>> for i in range(2,n+1) :
...     u = u1 + u2
...     u2 = u1
...     u1 = u
...
>>> tmp = u
>>> del n, u, u1, u2, i
>>> y = tmp
>>> del tmp
>>> y
55
```

On commence par copier la valeur du paramètre effectif `x` dans le paramètre formel d'entrée `n` (`n = x`) ; en PYTHON, `n` est vraiment créée à ce moment là seulement. On exécute ensuite « tel quel » le code de la fonction `fibonacci` : les variables internes à la fonction (`u`, `u1`, `u2`, `i`) sont créées et manipulées conformément à la définition de `fibonacci`. L'instruction « `return u` »

TD 3.6 : PASSAGE PAR VALEUR
On considère les codes suivants :

```
>>> x, y
(1, 2)
>>> tmp = x
>>> x = y
>>> y = tmp
>>> x, y
(2, 1)
```

```
def swap(x,y) :
    tmp = x
    x = y
    y = tmp
    return
```

```
>>> x, y
(1, 2)
>>> swap(x,y)
>>> x, y
(1, 2)
```

Expliquer la différence entre l'exécution de gauche et l'exécution de droite en explicitant l'appel équivalent à l'appel `swap(x,y)` *dans l'exécution de droite.*

provoque ensuite la création d'une variable temporaire `tmp` dans laquelle on copie la valeur du paramètre formel de sortie `u` (`tmp = u`) et toutes les variables introduites dans le corps de la fonction sont détruites (`del n, u, u1, u2, i`). On affecte la valeur de `tmp` à `y` (`y = tmp`) : cette affectation réalise effectivement l'affectation souhaitée (`y = fibonacci(x)`). Enfin, la variable temporaire `tmp` introduite au « retour de la fonction » `fibonacci` est détruite (`del tmp`). A la fin, seules coexistent les deux variables `x` et `y` : toutes les autres variables (`n`, `u`, `u1`, `u2`, `i` et `tmp`) ont été créées temporairement pour les besoins de l'appel de la fonction, puis détruites.

Définition 3.13 : passage par valeur
Le passage des paramètres par valeur (ou par copie) consiste à copier la valeur du paramètre effectif dans le paramètre formel correspondant.

Dans certains cas, le paramètre effectif occupe une trop grande place en mémoire pour envisager de le recopier. Par exemple, une image satellite de 6000×6000 pixels codés sur 8 niveaux de gris occupe $36Mo$ en mémoire ; la copier conduirait à occuper $72Mo$! Par ailleurs, on peut vouloir effectuer un traitement sur l'image originale et non traiter une de ses copies. Pour remedier à ces problèmes liés au passage des paramètres par valeur, on utilisera le « passage par référence » qui consiste à transférer la référence de la variable (son nom ou son adresse en mémoire) plutôt que sa valeur.

Définition 3.14 : passage par référence
Le passage des paramètres par référence consiste à copier la référence du paramètre effectif dans le paramètre formel correspondant.

En Python, certains types de variables utilisent systématiquement le passage par référence : c'est le cas des listes (type `list`), des dictionnaires (type `dict`) ou encore des classes (types définis par le programmeur).

```
def f(x):
    assert type(x) is list
    if len(x) > 0: x[0] = 5
    return
```

```
>>> t = [0,1,2]
>>> f(t)
>>> t
[5, 1, 2]
```

```
def g(x):
    assert type(x) is list
    y = [9,8,7]
    x = y
    x[0] = 5
    return
```

```
>>> t = [0,1,2]
>>> g(t)
>>> t
[0, 1, 2]
```

Les fonctions `f()` et `g()` nécessitent chacune un argument de type `list` (`assert type(x) is list`). Cet argument sera donc passé par référence lors de l'appel de ces fonctions : à l'intérieur de chacune d'elles, on manipulera une copie de la référence de la liste passée en argument (`t` dans les exemples de droite). La fonction `f()` cherche à modifier le premier élément de la liste (`x[0] = 5`) : elle affecte au premier élément de la liste nommée `x` (l'élément de rang `0` : `x[0]`) la valeur `5`. Comme le paramètre formel `x` est une copie de la référence originale `t`, il référence le même tableau de valeurs `[0,1,2]` et la modification portera bien sur l'élément `0` du tableau original. A la sortie de la fonction `f()`, la liste `t` originale est bien modifiée. Quant à elle, la fonction `g()` cherche à modifier la référence de la liste `x` (`x = y`). En fait, elle modifie la copie de la référence (`x`) mais pas la référence originale (`t`). L'affectation suivante (`x[0] = 5`) modifie ainsi le premier élément du tableau de valeurs `[9,8,7]`. Mais, à la sortie de la fonction `g()`, le tableau de valeurs `[0,1,2]` référencé par la liste `t` originale n'est pas modifié. Les mêmes fonctionnements sont obtenus avec un dictionnaire comme le montrent les exemples ci-dessous.

```
def f(x):
    assert type(x) is dict
    x[0] = 5
    return
```

```
>>> d = {}
>>> f(d)
>>> d
{0: 5}
```

```
def g(x):
    assert type(x) is dict
    y = {0: 9, 1: 8, 2: 7}
    x = y
    x[0] = 5
    return
```

```
>>> d = {}
>>> g(d)
>>> d
{}
```

3.3.2 Paramètres par défaut

Dans la définition d'une fonction, il est possible de définir une valeur par défaut pour chacun des paramètres d'entrée. On peut définir une valeur par défaut pour tous les paramètres, ou une partie d'entre eux seulement ; dans ce cas, les paramètres sans valeur par défaut doivent précéder les autres dans la liste. On obtient ainsi une fonction qui peut être appelée avec tous ses paramètres ou une partie seulement des arguments attendus :

```
def f(x, y = 0, z = 'r'):
    return x,y,z

>>> f(1,2)
(1, 2, 'r')
>>> f(1)
(1, 0, 'r')
```

```
>>> f(1,2,'t')
(1, 2, 't')

>>> f()
Traceback ...
TypeError: f() takes at least 1 argument
           (0 given)
```

De nombreuses fonctions des bibliothèques standards possèdent des arguments par défaut. C'est le cas par exemple de la fonction standard `int()` (voir annexe 3.5.2 page 153) ou encore de la fonction `log()` du module `math` (voir figure 3.3 page 102).

```
>>> int('45')
45
>>> int('45',10)
45
>>> int('101101',2)
45
>>> int('45',23)
97
>>> int('45',23) == 4*23**1 + 5*23**0
True
```

```
>>> from math import *
>>> e
2.7182818284590451
>>> log(e)
1.0
>>> log(e,e)
1.0
>>> log(e,10)
0.43429448190325176
>>> log(pi,7) == log(pi)/log(7)
True
```

A gauche, la fonction `int()` permet de transformer en un entier décimal une chaîne de caractères (par exemple `'45'`) qui représente un nombre écrit dans la base spécifiée par le 2ème argument (par exemple `23`). Par défaut, cette base est la base décimale (`10`). A droite, la fonction `log()` calcule le logarithme d'un nombre (par exemple `2.7182818284590451`) dans la base spécifiée par le 2ème argument (par exemple `10`). Par défaut, cette base est celle des logarithmes népériens (`e`). ■TD3.7

TD 3.7 : Valeurs par défaut
On considère la fonction qui code en base b sur k chiffres un nombre décimal n (voir TD 3.4). Proposer une définition de cette fonction où $k = 8$ et $b = 2$ par défaut.

Dans la plupart des langages de programmation, les arguments que l'on transmet à l'appel d'une fonction doivent être passés exactement dans le même ordre que celui des paramètres qui leur correspondent dans la définition de la fonction. En Python, si les paramètres annoncés

dans la définition de la fonction ont reçu chacun une valeur par défaut, on peut faire appel à la fonction en fournissant les arguments correspondants dans n'importe quel ordre, à condition de désigner nommément les paramètres correspondants.

```
def f(x = 0, y = 1):
    return x,y
>>> f()
(0, 1)
>>> f(3)
(3, 1)
```

```
>>> f(3,4)
(3, 4)
>>> f(y = 4, x = 3)
(3, 4)
>>> f(y = 6)
(0, 6)
>>> f(x = 8)
(8, 1)
```

La fonction `f()` a 2 paramètres d'entrée (`x` et `y`) qui admettent chacun une valeur par défaut (respectivement `0` et `1`). On peut l'appeler classiquement avec 0, 1 ou 2 arguments comme dans les 3 appels de gauche (`f()`, `f(3)`, `f(3,4)`). On peut aussi l'appeler en nommant explicitement les arguments comme dans les 3 exemples de droite (`f(y = 4, x = 3)`, `f(y = 6)`, `f(x = 8)`) ; dans ce cas, l'ordre des paramètres n'a plus d'importance.

```
>>> def f(x = 0, y = 1):
...     return x,y
...
```

```
>>> x = 7
>>> f(x = x)
(7, 1)
```

Dans ce dernier exemple, comment distinguer le paramètre formel `x` et le paramètre effectif `x` dans l'appel `f(x = x)` ? C'est le problème de la portée des variables.

Remarque 3.5 : *En* PYTHON*, l'espace de noms d'une fonction peut être connu grâce à deux fonctions standards :* `dir` *et* `locals` *(voir annexe 3.5.2 page 153). La fonction* `dir` *retourne la liste des variables locales à la fonction. La fonction* `locals` *retourne le dictionnaire « variable :valeur » des variables locales.*

```
>>> def f(x) :
...     y = 3
...     x = x + y
...     print('liste :', dir())
...     print('intérieur :', locals())
...     print('extérieur :', globals())
...     return x
...
>>> y = 6
>>> f(6)
liste : ['x', 'y']
intérieur : {'y' : 3, 'x' : 9}
extérieur : {'f' : <function f at 0x822841c>,
             'y' : 6,
             '__name__' : '__main__',
             '__doc__' : None}
9
```

La fonction standard `globals` *retourne le dictionnaire des variables globales. On constate qu'il y a bien 2 variables* `y`*, l'une globale qui vaut* `6` *et l'autre, locale, qui vaut* `3`*.*

3.3.3 Portée des variables

Les noms des paramètres effectifs n'ont ainsi rien de commun avec les noms des paramètres formels correspondants : il ne représente pas la même variable en mémoire... même s'ils ont le même nom !

Lorsqu'on définit des variables à l'intérieur du corps d'une fonction, ces variables ne sont accessibles qu'à la fonction elle-même. On dit que ces variables sont des « variables locales » à la fonction : en quelque sorte, elles sont confinées à l'intérieur de la fonction. C'est par exemple le cas des variables `n`, `u`, `u1`, `u2` et `i` de la fonction `fibonacci` (voir section 3.2.6). Chaque fois que la fonction `fibonacci` est appelée, PYTHON réserve pour elle en mémoire un nouvel « espace de noms ». Les valeurs des variables locales `n`, `u`, `u1`, `u2` et `i` sont ainsi stockées dans cet espace de noms qui est inaccessible depuis l'extérieur de la fonction. Ainsi par exemple, si nous essayons d'afficher le contenu de la variable `u` juste après avoir effectué un appel à la fonction `fibonacci`,

on obtient un message d'erreur (`name 'u' is not defined`).

```
>>> from fibo import fibonacci
>>> fibonacci(9)
55
>>> u
Traceback ...
NameError: name 'u' is not defined
```

L'espace de noms réservé lors de l'appel de la fonction est automatiquement détruit dès que la fonction a terminé son travail (voir les appels équivalents en section 3.3.1).

Les variables définies à l'extérieur d'une fonction sont des « variables globales ». Leur valeur est « visible » de l'intérieur d'une fonction, mais la fonction ne peut pas *a priori* la modifier.

```
>>> def f(x):
...     y = 3
...     x = x + y
...     return x
...
>>> y = 6
>>> f(1)
4
>>> y
6
```

On constate que la variable `y` n'a pas changé de valeur entre « avant » et « après » l'appel à la fonction `f` bien qu'il semble qu'elle soit modifiée à l'intérieur de `f` (`y = 3`). En fait, il ne s'agit pas des mêmes variables : l'une est globale (extérieur à la fonction) et vaut `6`, l'autre est locale (intérieur à la fonction) et vaut `3`.

Si deux variables portent le même nom à l'extérieur et à l'intérieur d'une fonction, la priorité est donnée à la variable locale à l'intérieur de la fonction ; à l'extérieur de la fonction, le problème ne se pose pas : la variable locale n'existe pas. ■TD3.8

TD 3.8 : Portée des variables

On considère les fonctions `f`, `g` *et* `h` *suivantes :*

```
def f(x) :          def g(x) :          def h(x) :
  x = 2*x             x = 2*f(x)          x = 2*g(f(x))
  print('f', x)       print('g', x)       print('h', x)
  return x            return x            return x
```

Qu'affichent les appels suivants ?

1.
```
>>> x = 5
>>> print(x)
?
>>> y = f(x)
>>> print(x)
?
>>> z = g(x)
>>> print(x)
?
>>> t = h(x)
>>> print(x)
?
```

2.
```
>>> x = 5
>>> print(x)
?
>>> x = f(x)
>>> print(x)
?
>>> x = g(x)
>>> print(x)
?
>>> x = h(x)
>>> print(x)
?
```

3.3.4 Appels récursifs

La suite des nombres u_n de Fibonacci est définie par la relation de récurrence suivante (exemple 3.4 page 104) :

$$u_0 = 1\ ,\ u_1 = 1\ ,\ u_n = u_{n-1} + u_{n-2}\ \forall n \in N,\ n > 1$$

La relation de récurrence exprime le nombre u_n à l'ordre n en fonction des nombres u_{n-1} et u_{n-2}, respectivement à l'ordre $(n-1)$ et $(n-2)$. Ainsi, on définit u_n directement en fonction de u_{n-1} et de u_{n-2}. Cette écriture est très compacte et très expressive ; en particulier, elle ne fait pas intervenir de boucle comme dans le code de la fonction définie en section 3.2.6. Il

serait donc intéressant de définir la fonction `fibonacci` de manière aussi simple dans le langage algorithmique.

Version itérative :

```
def fibonacci(n) :
    u, u1, u2 = 1, 1, 1
    for i in range(2,n+1) :
        u = u1 + u2
        u2 = u1
        u1 = u
    return u
```

Version récursive :

```
def fibonacci(n) :
    u = 1
    if n > 1 :
        u = fibonacci(n-1) + fibonacci(n-2)
    return u
```

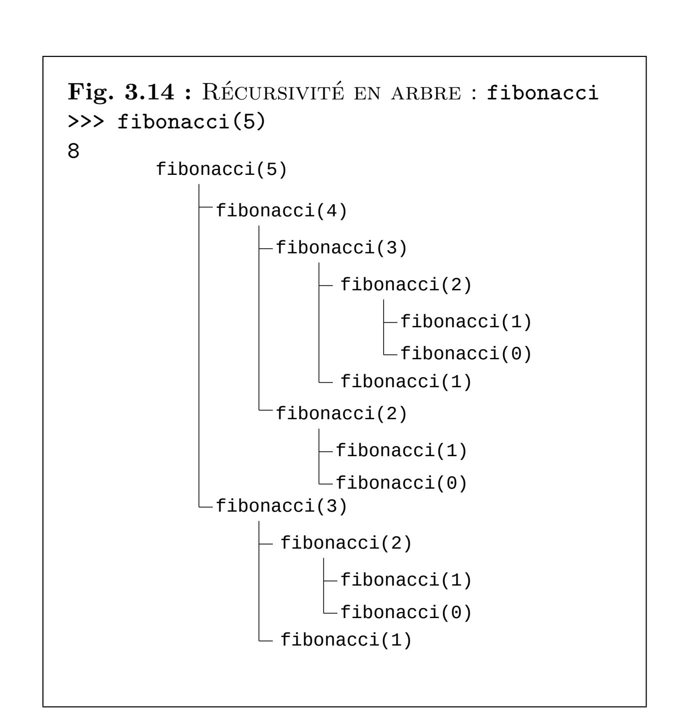
On retrouve à gauche la version itérative déjà proposée en section 3.2.6. A droite, la version est dite « récursive » parce que dans le corps de la fonction `fibonacci`, on appelle la fonction `fibonacci` elle-même, et on l'appelle même 2 fois. Cette version récursive est la traduction directe de la formulation mathématique.

Définition 3.15 : FONCTION RÉCURSIVE
Une fonction est dite récursive si elle s'appelle elle-même : on parle alors d'appel récursif de la fonction.

Dans la version récursive, pour calculer `fibonacci(5)`, on calcule d'abord `fibonacci(4)` et `fibonacci(3)`. Pour calculer `fibonacci(4)`, on calcule `fibonacci(3)` et `fibonacci(2)`. Pour calculer `fibonacci(3)`, on calcule `fibonacci(2)` et `fibonacci(1)`... Le déroulement du processus ressemble ainsi à un arbre (figure 3.14). On remarque que les branches de l'arbre se divise en deux à chaque niveau (sauf en bas de l'arbre, *ie* à droite sur la figure), ce qui traduit le fait que la fonction `fibonacci` s'appelle elle-même deux fois à chaque fois qu'elle est invoquée avec `n > 1`. Dans cet arbre, on constate par exemple que le calcul de `fibonacci(3)` est développé intégralement 2 fois : une fois pour le calul de `fibonacci(4)` et une fois pour lui-même. En fait, il n'est pas très difficile de montrer que le nombre de fois où la fonction calcule `fibonacci(1)` ou `fibonacci(0)` (*ie.* le nombre de feuilles dans l'arbre) est précisément u_n (`fibonacci(n)`). Or la valeur de u_n croît de manière exponentielle avec n ; ainsi, avec cette version récursive, le processus de calcul de `fibonacci(n)` prend un temps qui croît de façon exponentielle avec `n`.

Dans la version itérative, on ne passe que $(n-1)$ fois dans la boucle. Ainsi, le processus de calcul itératif de `fibonacci(n)` prend un temps qui croît de manière linéaire avec `n`. La différence entre les temps de calcul requis par les 2 méthodes, l'une linéaire en n et l'autre augmentant aussi vite que u_n lui-même, est donc énorme même pour de petites valeurs de n. Par exemple, pour $n = 50$, il faudra 50 unités de temps pour la méthode itérative contre 20365011074 (plus de

Fig. 3.14 : RÉCURSIVITÉ EN ARBRE : `fibonacci`



20 milliards unités de temps !) pour la méthode récursive. La version itérative sera donc préférée à la version récursive dans ce cas là : « il n'y a pas photo » !

Il ne faudrait pas conclure à l'inutilité des processus récursifs en arbre. Pour les processus opérant sur des structures de données hiérarchiques et non plus sur des nombres, la récursivité en arbre est un outil naturel et puissant[1]. Même pour les calculs numériques, les processus récursifs en arbre peuvent être utiles à la compréhension et à la conception d'algorithmes. Par exemple, bien que la version récursive de `fibonacci` soit beaucoup moins efficace que la version itérative, elle s'obtient presque directement, étant à peine plus qu'une traduction en PYTHON de la définition mathématiques des nombres de Fibonacci. En revanche, pour formuler la version itérative, il fallait avoir remarqué que le calcul pouvait être revu sous la forme d'une itération avec 3 variables ; ce qui est bien moins direct et moins intuitif que la version récursive.

Exemple 3.5 : TOURS DE HANOÏ

Les « tours de Hanoï » est un jeu imaginé par le mathématicien français Édouard Lucas (1842-1891). Il consiste à déplacer n disques de diamètres différents d'une tour de « départ » à une tour d'« arrivée » en passant par une tour « intermédiaire » et ceci en un minimum de coups, tout en respectant les règles suivantes :

- *on ne peut déplacer qu'un disque à la fois,*
- *on ne peut placer un disque que sur un autre disque plus grand que lui ou sur une tour vide.*

Dans l'état initial, les n disques sont placés sur la tour « départ ». Dans l'état final, tous les disques se retrouvent placés dans le même ordre sur la tour « arrivéee » (figure 3.15). ■TD3.9

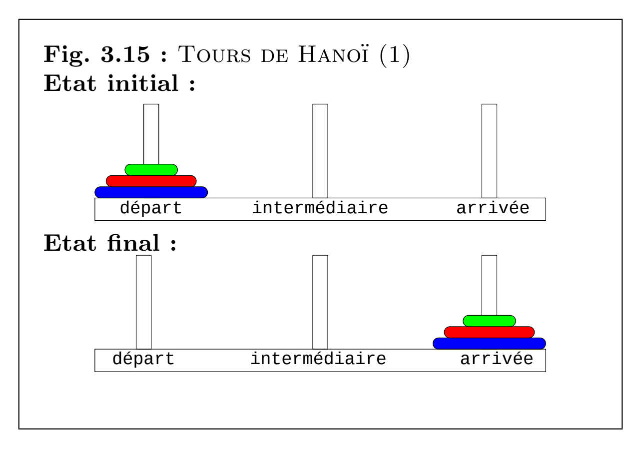


Fig. 3.15 : TOURS DE HANOÏ (1)
Etat initial :
Etat final :

TD 3.9 : TOURS DE HANOÏ *à la main*
Résoudre à la main *le problème des tours de Hanoï à n disques (voir exemple 3.5 et figure 3.15) successivement pour $n = 1$, $n = 2$, $n = 3$ et $n = 4$.*

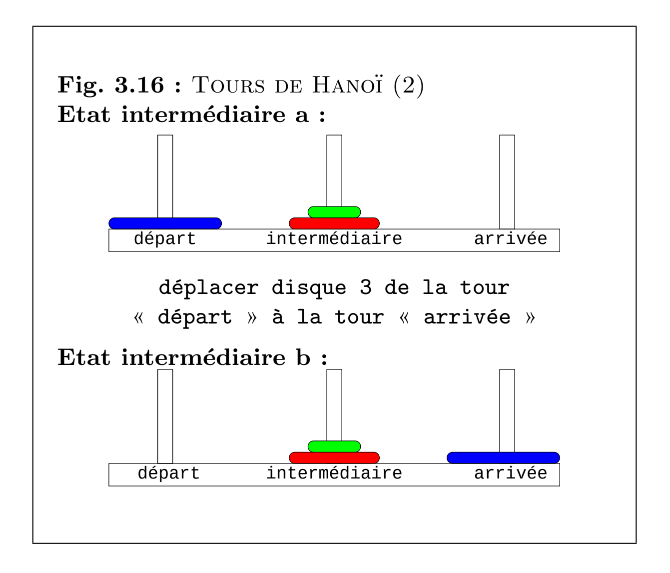


Fig. 3.16 : TOURS DE HANOÏ (2)
Etat intermédiaire a :
Etat intermédiaire b :

On cherche donc à définir une procédure `hanoi(n,depart,intermediaire,arrivee)` qui devra déplacer `n` disques de la tour `depart` à la tour `arrivee` en utilisant la tour `intermediaire` comme tour de transit. Numérotons $1, 2, 3, \ldots, n$ les n disques du plus petit (numéro 1) au plus grand (numéro n). A un moment donné, dans la suite des opérations à effectuer, il faudra déplacer le disque numéro n (le plus grand, placé initialement en dessous de la pile de disques) de la tour « départ » à la tour « arrivée ». Pour pouvoir effectuer ce déplacement, il faut d'une part qu'il n'y ait plus aucun disque sur le disque n et d'autre part que la tour « arrivée » soit vide ; en conséquence, il faut que tous les autres disques (de 1 à $(n - 1)$) soient sur la tour « intermédiaire » (figure 3.16 a). Pour atteindre cet état intermédiaire, il faut

[1] L'étude des structures de données arborescentes est abordée à partir du 5ème semestre des enseignements d'informatique à l'ENIB.

donc déplacer les $(n-1)$ premiers disques de la tour « départ » à la tour « intermédiaire » en utilisant la tour « arrivée » comme tour de transit : ce déplacement correspond à l'appel `hanoi(n-1,depart,arrivee,intermediaire)`. Une fois réalisé ce déplacement des $(n-1)$ premiers disques, le disque n peut être déplacé de la tour « départ » à la tour « arrivée » (figure 3.16 b). Il ne reste plus qu'à déplacer les $(n-1)$ premiers disques de la tour « intermédiaire » à la tour « arrivée » en utilisant la tour « départ » comme tour de transit ; ces derniers déplacements correspondent à l'appel `hanoi(n-1,intermediaire,depart,arrivee)`. On en déduit l'implémentation de la procédure `hanoi` de la page suivante où le déplacement y est traduit par un simple affichage du type : `déplacer disque 3 de la tour « départ » à la tour « arrivée »`. Un appel de cette procédure pour $n = 3$ est donné à titre d'exemple ; les tours « départ », « intermédiaire » et « arrivée » y sont respectivement nommées `'d'`, `'i'` et `'a'`.

```
def hanoi(n,gauche,milieu,droit):
    assert type(n) is int
    assert n >= 0
    if n > 0:
        hanoi(n-1,gauche,droit,milieu)
        deplacer(n,gauche,droit)
        hanoi(n-1,milieu,droit,gauche)
    return

def deplacer(n,gauche,droit):
    print('déplacer disque', n,
          'de la tour', gauche,
          'à la tour', droit)
    return
```

```
>>> hanoi(3,'d','i','a')
déplacer disque 1 de la tour d à la tour a
déplacer disque 2 de la tour d à la tour i
déplacer disque 1 de la tour a à la tour i
déplacer disque 3 de la tour d à la tour a
déplacer disque 1 de la tour i à la tour d
déplacer disque 2 de la tour i à la tour a
déplacer disque 1 de la tour d à la tour a
```

L'exécution d'un appel à la procédure `hanoi` s'apparente ici encore à un processus récursif en arbre : les 7 déplacements effectués lors de l'appel `hanoi(3,'d','i','a')` sont numérotés dans leur ordre d'apparition sur la figure 3.17 (les appels à la fonction `hanoi` pour `n = 0` ne font rien). Mais toutes les fonctions récursives ne conduisent pas nécessairement à un processus récursif en arbre comme l'exemple de la fonction `factorielle` le montre.

Fig. 3.17 : Récursivité en arbre : `hanoi`

```
>>> hanoi(3,'d','i','a')

hanoi(3,'d','i','a')
 ├─ hanoi(2,'d','a','i')
 │   ├─ hanoi(1,'d','i','a')
 │   │   ├─ hanoi(0,'d','a','i')
 │   │   ├─ deplacer(1,'d','a')        1
 │   │   └─ hanoi(0,'i','d','a')
 │   ├─ deplacer(2,'d','i')            2
 │   └─ hanoi(1,'a','d','i')
 │       ├─ hanoi(0,'a','i','d')
 │       ├─ deplacer(1,'a','i')        3
 │       └─ hanoi(0,'d','a','i')
 ├─ deplacer(3,'d','a')                4
 └─ hanoi(2,'i','d','a')
     ├─ hanoi(1,'i','a','d')
     │   ├─ hanoi(0,'i','d','a')
     │   ├─ deplacer(1,'i','d')        5
     │   └─ hanoi(0,'a','i','d')
     ├─ deplacer(2,'i','a')            6
     └─ hanoi(1,'d','i','a')
         ├─ hanoi(0,'d','a','i')
         ├─ deplacer(1,'d','a')        7
         └─ hanoi(0,'i','d','a')
```

Exemple 3.6 : Fonction factorielle

La fonction factorielle qui calcule le produit des n premiers entiers positifs ($n! = \prod_{k=1}^{n} k$) *est*

simplement définie par la relation de récurrence :

$$\begin{cases} 0! = 1 \\ n! = n \cdot (n-1)! \quad \forall n \in N^* \end{cases}$$

Version itérative :

```
def factorielle(n) :
    u = 1
    for i in range(2,n+1) :
        u = u * i
    return u
```

Version récursive :

```
def factorielle(n) :
    u = 1
    if n > 1 :
        u = n * factorielle(n-1)
    return u
```

On retrouve à gauche la version itérative bien connue. A droite, la version récursive est la traduction directe de la formulation mathématique. Dans la version récursive, le processus nécessite que l'interpréteur garde une trace des multiplications à réaliser plus tard (figure 3.18). Le processus croît puis décroît : la croissance se produit lorsque le processus construit une chaîne d'opérations différées (ici, une chaîne de multiplications différées) et la décroissance intervient lorsqu'on peut évaluer les multiplications. Ainsi, la quantité d'information qu'il faut mémoriser pour effectuer plus tard les opérations différées croît linéairement avec n : on parle de processus récursif linéaire. L'interprétation d'une fonction récursive passe donc par une phase d'expansion dans lesquels les appels récursifs sont « empilés » jusqu'à arriver à un appel de la fonction pour lequel une condition d'arrêt sera vérifiée, puis par une phase de contraction dans laquelle les résultats des appels précédemment empilés sont utilisés. Par contre, dans la version itérative, le processus de calcul ne croît ni ne décroît : à chaque étape, seules les valeurs courantes des variables `u` et `i` sont nécessaires (il n'y a pas d'opérations différées) et le temps requis pour calculer $n!$ augmente linéairement avec n (on passe $n-2$ fois dans la boucle). ■ TD3.10

Lorsqu'on parle de fonction récursive, on fait référence à une caractéristique syntaxique : la fonction, dans sa propre définition, se fait réfèrence à elle-même (elle s'appelle elle-même). Mais lorsqu'on parle de processus récursif, linéaire ou en arbre, on s'intéresse au déroulement du processus, et non à la syntaxe de l'écriture de la fonction. Ainsi, une fonction peut avoir une définition récursive mais correspondre à un processus itératif : c'est le cas de la nouvelle version de la fonction `factorielle` ci-dessous.

Fig. 3.18 : Récursivité linéaire : `factorielle`

```
factorielle(5)
(5*factorielle(4))
(5*(4*factorielle(3)))
(5*(4*(3*factorielle(2))))
(5*(4*(3*(2*factorielle(1)))))
(5*(4*(3*(2*1))))
(5*(4*(3*2)))
(5*(4*6))
(5*24)
120
```

TD 3.10 : Pgcd et ppcm de 2 entiers (1)

1. *Définir une fonction récursive qui calcule le plus grand commun diviseur d de 2 entiers a et b :*
   $\mathrm{pgcd(a,b) = pgcd(b, a \bmod b)} = \ldots$
   $\ldots = \mathrm{pgcd(d,0) = d}$.
2. *En déduire une fonction qui calcule le plus petit commun multiple m de 2 entiers a et b.*

```
def factorielle(n):
    u = factIter(n,1,1)
    return u

def factIter(n,i,fact):
    u = fact
    if i < n:
        u = factIter(n,i+1,fact*(i+1))
    return u
```

```
factorielle(5)
(factIter(5,1,1))
(factIter(5,2,2))
(factIter(5,3,6))
(factIter(5,4,24))
(factIter(5,5,120))
120
```

La nouvelle fonction `factorielle` appelle une fonction auxiliaire `factIter` dont la définition est syntaxiquement récursive (`factIter` s'appelle elle-même). Cette fonction à 3 arguments : l'entier `n` dont il faut calculer la factorielle, un compteur `i` initialisé à `1` au premier appel de `factIter` par `factorielle` et incrémenté à chaque nouvel appel, et un nombre `fact` initialisé à `1` et multiplié par la nouvelle valeur du compteur à chaque nouvel appel. Le déroulement d'un appel à `factIter` montre qu'ainsi, à chaque étape, la relation `(i ! == fact)` est toujours vérifiée. La fonction `factIter` arrête de s'appeler elle-même lorsque `(i == n)` et on a alors `(fact == i ! == n !)` qui est la valeur recherchée. Ainsi, à chaque étape, nous n'avons besoin que des valeurs courantes du compteur `i` et du produit `fact`, exactement comme dans la version itérative de la fonction `factorielle` : il n'y a plus de chaîne d'opérations différées comme dans la version récursive de `factorielle`. Le processus mis en jeu ici est un processus itératif, bien que la définition de `factIter` soit récursive. ■ TD3.11

TD 3.11 : Somme arithmétique

1. *Définir une fonction récursive qui calcule la somme des n premiers nombres entiers.*

$$s = \sum_{k=0}^{n} k = \frac{n(n+1)}{2}$$

2. *Comparer la complexité de cette version avec les versions constante et itérative (voir TD 3.5).*

Dans la fonction `factIter`, le résultat de l'appel récursif est retourné par la fonction : on parle alors de récursivité terminale (ou récursivité à droite). L'exécution d'un tel appel termine l'exécution de la fonction.

Définition 3.16 : récursivité terminale

Un appel récursif terminal est un appel récursif dont le résultat est celui retourné par la fonction.

En d'autres termes, si dans le corps d'une fonction, un appel récursif est placé de telle façon que son exécution n'est jamais suivi par l'exécution d'une autre instruction de la fonction, cet appel est dit récursif à droite.

Définition 3.17 : récursivité non terminale

Un appel récursif non terminal est un appel récursif dont le résultat n'est pas celui retourné par la fonction.

■ TD3.12

TD 3.12 : Courbes fractales

On s'intéresse ici aux programmes dont l'exécution produit des dessins à l'aide de la tortue Logo (voir annexe 3.5.1 page 152). On considère la procédure `draw` *ci-dessous :*

```
def draw(n,d) :
    assert type(n) is int
    assert n >= 0
    if n == 0 : forward(d)
    else :
        draw(n-1,d/3.)
        left(60)
        draw(n-1,d/3.)
        right(120)
        draw(n-1,d/3.)
        left(60)
        draw(n-1,d/3.)
    return
```

Dessiner le résultat des appels `draw(n,900)` *respectivement pour* `n = 0`, `n = 1`, `n = 2` *et* `n = 3`. *A chaque appel, le crayon est initialement en* $(0,0)$ *avec une direction de* 0.

Quel que soit le problème à résoudre, on a le choix entre l'écriture d'une fonction itérative et celle d'une fonction récursive. Si le problème admet une décomposition récurrente naturelle, le programme récursif est alors une simple adaptation de la décomposition choisie. C'est le cas des fonctions `factorielle` et `fibonacci` par exemple. L'approche récursive présente cependant des inconvénients : certains langages n'admettent pas la récursivité (comme le langage machine !) et elle est souvent coûteuse en mémoire comme en temps d'exécution. On peut pallier ces inconvénients en transformant la fonction récursive en fonction itérative : c'est toujours possible.

Considérons une procédure `f` à récursivité terminale écrite en pseudo-code :

```
def f(x):
    if cond: arret
    else:
        instructions
        f(g(x))
    return
```

`x` représente ici la liste des arguments de la fonction, `cond` une condition portant sur `x`, `instructions` un bloc d'instructions qui constituent le traitement de base de la fonction `f`, `g(x)` une transformation des arguments et `arret` l'instruction de terminaison (clause d'arrêt) de la récurrence.

Elle est équivalente à la procédure itérative suivante :

```
def f(x):
    while not cond:
        instructions
        x = g(x)
    arret
    return
```

Illustrons cette transformation à l'aide de la fonction qui calcule le `pgcd` de 2 entiers.

```
def pgcd(a,b):
    if b == 0: return a
    else:
        pass # ne fait rien
        return pgcd(b,a%b)

>>> pgcd(12,18)
6
```

`x`	→ `a,b`
`cond`	→ `b == 0`
`arret`	→ `return a`
`instructions`	→ pass
`x = g(x)`	→ `a,b = b,a%b`

```
def pgcd(a,b):
    while not (b == 0):
        pass
        a,b = b,a%b
    return a

>>> pgcd(12,18)
6
```

La méthode précédente ne s'applique qu'à la récursivité terminale. Une méthode générale existe pour transformer une fonction récursive quelconque en une fonction itérative équivalente. En particulier, elle est mise en œuvre dans les compilateurs car le langage machine n'admet pas la récursivité. Cette méthode générale fait appel à la notion de pile (section 4.2.4 page 167) pour sauvegarder le contexte des appels récursifs. On l'illustrera dans le cas particulier du tri d'une liste par la méthode du tri rapide (section 4.4.3 page 177).

3.4 Exercices complémentaires

3.4.1 Connaître

TD 3.13 : QCM (3)

(un seul item correct par question)

1. *La réutilisabilité d'un algorithme est son aptitude*
   (a) *à utiliser de manière optimale les ressources du matériel qui l'exécute*
   (b) *à se protéger de conditions anormales d'utilisation*
   (c) *à résoudre des tâches équivalentes à celle pour laquelle il a été conçu*
   (d) *à réaliser exactement la tâche pour laquelle il a été conçu*
2. *L'encapsulation est l'action*
   (a) *de mettre une chose dans une autre*
   (b) *de fermer une chose par une autre*
   (c) *de substituer une chose par une autre*
   (d) *de remplacer une chose par une autre*
3. *Une fonction est un bloc d'instructions nommé et paramétré*
   (a) *qui ne peut pas retourner plusieurs valeurs*
   (b) *qui ne peut pas contenir d'instructions itératives*
   (c) *qui retourne une valeur*
   (d) *qui ne retourne pas de valeur*
4. *Les paramètres d'entrée d'une fonction sont*
   (a) *les arguments nécessaires pour effectuer le traitement associé à la fonction*
   (b) *les valeurs obtenues après avoir effectué le traitement associé à la fonction*
   (c) *des grandeurs invariantes pendant l'exécution de la fonction*
   (d) *des variables auxiliaires définies dans le corps de la fonction*
5. *Les préconditions d'une fonction sont des conditions à respecter*
   (a) *par les paramètres de sortie de la fonction*

   (b) *pendant toute l'exécution de la fonction*
   (c) *par les paramètres d'entrée de la fonction*
   (d) *pour pouvoir compiler la fonction*

6. *La description d'une fonction décrit*
   (a) *ce que fait la fonction*
   (b) *comment fait la fonction*
   (c) *pourquoi la fonction le fait*
   (d) *où la fonction le fait*

7. *Le jeu de tests d'une fonction est*
   (a) *un ensemble d'exercices à résoudre*
   (b) *un ensemble d'exceptions dans le fonctionnement de la fonction*
   (c) *un ensemble caractéristiques d'entrées-sorties associées*
   (d) *un ensemble de recommandations dans l'utilisation de la fonction*

8. *En* PYTHON, *l'instruction* `assert` *permet de*
   (a) *tester une précondition*
   (b) *imposer une instruction*
   (c) *paramétrer une fonction*
   (d) *tester un test du jeu de tests*

9. *La validité d'une fonction est son aptitude à réaliser exactement la tâche pour laquelle elle a été conçue. Plus concrètement,*
   (a) *la fonction doit vérifier impérativement ses préconditions*
   (b) *la fonction doit être correctement paramétrée*
   (c) *l'implémentation de la fonction doit être conforme aux jeux de tests*
   (d) *l'utilisation de la fonction doit être conviviale*

10. *Le passage des paramètres par valeur consiste à copier*
   (a) *la valeur du paramètre formel dans le paramètre effectif correspondant*
   (b) *la référence du paramètre effectif dans le paramètre formel correspondant*

   (c) *la référence du paramètre formel dans le paramètre effectif correspondant*

   (d) *la valeur du paramètre effectif dans le paramètre formel correspondant*

11. *Un appel récursif est un appel*

   (a) *dont l'exécution est un processus récursif*

   (b) *dont l'exécution est un processus itératif*

   (c) *dont le résultat est retourné par la fonction*

   (d) *d'une fonction par elle-même*

3.4.2 Comprendre

TD 3.14 : Passage des paramètres
On considère les fonctions `f`, `g` *et* `h` *suivantes :*

```
def f(x):
    y = x + 2
    return y
```

```
def g(z):
    v = 2*f(z)
    return v
```

```
def h(a):
    b = g(f(a))
    return b
```

Quels sont les algorithmes équivalents (algorithmes où il n'y a plus d'appels aux fonctions `f`, `g` *et* `h`*) aux appels suivants :*

1. `u = f(2)`
2. `u = f(t)`
3. `u = g(2)`
4. `u = g(t)`
5. `u = h(2)`
6. `u = h(t)`

TD 3.15 : Portée des variables (2)
On considère les fonctions `f`, `g` *et* `h` *suivantes :*

```
def f(x):
    x = x + 2
    print('f', x)
    return x
```

```
def g(x):
    x = 2*f(x)
    print('g', x)
    return x
```

```
def h(x):
    x = g(f(x))
    print('h', x)
    return x
```

Qu'affichent les appels suivants ?

1.
```
>>> x = 5
>>> print(x)

>>> x = x + 2
>>> print(x)

>>> x = 2 * (x + 2)
>>> print(x)
```

2.
```
>>> x = 5
>>> print(x)

>>> y = f(x)
>>> print(x, y)

>>> z = 2*f(y)
>>> print(x, y, z)
```

3.
```
>>> x = 5
>>> print(x)

>>> z = 2*f(f(x))
>>> print(x, z)
```

4.
```
>>> x = 5
>>> print(x)

>>> f(x)

>>> print(x)

>>> g(x)

>>> print(x)

>>> h(x)

>>> print(x)
```

3.4.3 Appliquer

TD 3.16 : Suite géométrique
Définir une fonction récursive qui calcule la somme des n *premiers termes d'une suite géométrique* $u_k = ab^k$.

TD 3.17 : Puissance entière
Définir une fonction récursive qui calcule la puissance entière $p = x^n$ *d'un nombre entier* x.

TD 3.18 : Coefficients du binôme
Définir une fonction récursive qui calcule les coefficients du binôme $(a+b)^n = \sum_{k=0}^{n} \frac{n!}{k!(n-k)!} a^{n-k} b^k$.

TD 3.19 : Fonction d'Ackerman
Définir une fonction récursive qui calcule la fonction d'Ackerman :

$$f : N^2 \rightarrow N \left\{ \begin{array}{lcl} f(0,n) & = & n+1 \\ f(m,0) & = & f(m-1,1) \ si \ m > 0 \\ f(m,n) & = & f(m-1,f(m,n-1)) \ si \ m > 0, n > 0 \end{array} \right.$$

3.4.4 Analyser

TD 3.20 : Addition binaire
Définir une fonction `add2` *qui effectue l'addition binaire de 2 entiers a et b (le nombre de bits n'est pas limité* a priori*).*
Exemple : $(0101)_2 + (10011)_2 = (11000)_2$

```
# add2(a,b)
>>> add2([1,0],[1,0,1,1])
[1, 1, 0, 1]
>>> add2([1,0,1,1],[1,0])
[1, 1, 0, 1]
>>> add2([1,1],[1,1])
[1, 1, 0]
```

TD 3.21 : Complément à 2
Définir une fonction `neg2` *qui détermine le complément à 2 en binaire d'un entier n codé sur k bits.*

$$(011100)_2 \rightarrow (100100)_2 \ : \ \begin{array}{rr} & (011100)_2 \\ \hline & (100011)_2 \\ + & (000001)_2 \\ \hline = & (100100)_2 \end{array}$$

```
# neg2(code)
>>> neg2([0,0,0,1,0,1,1,1])
[1, 1, 1, 0, 1, 0, 0, 1]
>>> neg2([1, 1, 1, 0, 1, 0, 0, 1])
[0, 0, 0, 1, 0, 1, 1, 1]
>>> for a in [0,1]:
...     for b in [0,1]:
...         for c in [0,1]:
...             add2([a,b,c],neg2([a,b,c]))
[0, 0, 0]
[1, 0, 0, 0]
[1, 0, 0, 0]
[1, 0, 0, 0]
[1, 0, 0, 0]
[1, 0, 0, 0]
[1, 0, 0, 0]
[1, 0, 0, 0]
```



IEEE : `www.ieee.org`
Institute of Electrical and Electronics Engineers

TD 3.22 : Codage-décodage des réels

Remarque 3.6 : *On pourra vérifier les résultats obtenus avec la fonction* `ieee` *du TD 3.22 ci-contre sur le site* `http://babbage.cs.qc.edu/IEEE-754/Decimal.html`

1. *Définir une fonction* `ieee` *qui code un nombre réel x selon la norme IEEE 754 simple précision.*

$$x = (-1)^s \cdot (1 + m) \cdot 2^{(e-127)}$$

```
# ieee(x)
>>> ieee(0.0)
[0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
>>> ieee(0.625)
[0, 0, 1, 1, 1, 1, 1, 1, 0,
 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
>>> ieee(3.1399998664855957)
[0, 1, 0, 0, 0, 0, 0, 0, 0,
 1, 0, 0, 1, 0, 0, 0, 1, 1, 1, 1,
 0, 1, 0, 1, 1, 1, 0, 0, 0, 0, 1, 0]
>>> ieee(-4573.5)
[1, 1, 0, 0, 0, 1, 0, 1, 1,
 0, 0, 0, 1, 1, 1, 0, 1, 1, 1, 0,
 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

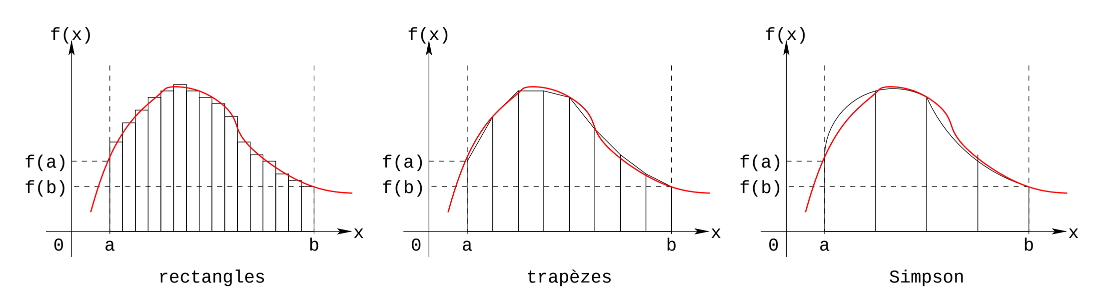
2. *Définir une fonction* `real` *qui décode un nombre réel x codé selon la norme IEEE 754 simple précision.*

$$x = (-1)^s \cdot (1+m) \cdot 2^{(e-127)}$$

```
# real(code)
>>> real(ieee(0.625))
0.625
>>> real(ieee(3.1399998664855957))
3.1399998664855957
>>> real(ieee(-4573.5))
-4573.5
```

TD 3.23 : Intégration numérique

Soit $f(x)$ une fonction continue de $R \to R$ à intégrer sur $[a, b]$ (on supposera que f à toutes les bonnes propriétés mathématiques pour être intégrable sur l'intervalle considéré). On cherche à calculer son intégrale $I = \int_a^b f(x)dx$ qui représente classiquement l'aire comprise entre la courbe représentative de f et les droites d'équations $x = a$, $x = b$ et $y = 0$. Les méthodes d'intégration numérique (méthode des rectangles, méthode des trapèzes et méthode de Simpson) consistent essentiellement à trouver une bonne approximation de cette aire.



On testera ces différentes méthodes avec la fonction $f(x) = \sin(x)$ sur $[0, \pi]$.

1. *Méthode des rectangles : subdivisons l'intervalle d'intégration de longueur $b-a$ en n parties égales de longueur $\Delta x = \dfrac{b-a}{n}$. Soient x_1, x_2, …, x_n les points milieux de ces n intervalles. Les n rectangles formés avec les ordonnées correspondantes ont pour surface $f(x_1)\Delta x$, $f(x_2)\Delta x$, …, $f(x_n)\Delta x$. L'aire sous la courbe est alors assimilée à la somme des aires de*

ces rectangles, soit

$$I = \int_a^b f(x)dx \approx (f(x_1) + f(x_2) + \cdots + f(x_n))\,\Delta x$$

C'est la formule dite des rectangles qui repose sur une approximation par une fonction en escalier.

Ecrire une fonction `rectangle_integration` *qui calcule l'intégrale définie I d'une fonction f sur $[a, b]$ à l'ordre n par la méthode des rectangles.*

2. *Méthode des trapèzes : subdivisons l'intervalle d'intégration de longueur $b-a$ en n parties égales de longueur $\Delta x = \dfrac{b-a}{n}$. Les abscisses des points ainsi définis sont a, x_1, x_2, ..., x_{n-1}, b et les trapèzes construits sur ces points et les ordonnées correspondantes ont pour aire $\dfrac{\Delta x}{2}(f(a) + f(x_1))$, $\dfrac{\Delta x}{2}(f(x_1) + f(x_2))$, ..., $\dfrac{\Delta x}{2}(f(x_{n-1}) + f(b))$. L'aire sous la courbe est alors assimilée à la somme des aires de ces trapèzes, soit*

$$I = \int_a^b f(x)dx \approx \left(\frac{f(a) + f(b)}{2} + f(x_1) + f(x_2) + \cdots + f(x_{n-1})\right)\Delta x$$

C'est la formule dite des trapèzes.

Ecrire une fonction `trapezoid_integration` *qui calcule l'intégrale définie I d'une fonction f sur $[a, b]$ à l'ordre n par la méthode des trapèzes.*

3. *Méthode de Simpson : divisons l'intervalle d'intégration $[a, b]$ en un nombre n pair d'intervalles dont la longueur est $\Delta x = \dfrac{b-a}{n}$. Dans les 2 premiers intervalles d'extrémités a, x_1 et x_2, on approche la courbe représentative de f par une parabole d'équation $y = \alpha x^2 + \beta x + \gamma$ passant par les points $A(a, f(a))$, $A_1(x_1, f(x_1))$ et $A_2(x_2, f(x_2))$ de la courbe. Dans les 2 intervalles suivants, on approche la courbe par une autre parabole d'équation similaire, passant par les points A_2, A_3 et A_4, et ainsi de suite. On obtient ainsi une courbe formée de n portions de parabole et l'aire déterminée par ces portions de parabole est une approximation de l'aire I cherchée.*

L'intégration de l'équation de la parabole $y = \alpha x^2 + \beta x + \gamma$ sur $[-\Delta x, \Delta x]$ donne

$$S = \int_{-\Delta x}^{\Delta x} (\alpha x^2 + \beta x + \gamma)dx = \frac{2}{3}\alpha(\Delta x)^3 + 2\gamma(\Delta x)$$

où les constantes α et γ sont déterminées en écrivant que les points $(-\Delta x, y_0)$, $(0, y_1)$ et $(\Delta x, y_2)$ satisfont l'équation de la parabole. On obtient ainsi :

$$\left|\begin{array}{l} y_0 = \alpha(-\Delta x)^2 + \beta(-\Delta x) + \gamma \\ y_1 = \gamma \\ y_2 = \alpha(\Delta x)^2 + \beta(\Delta x) + \gamma \end{array}\right. \Rightarrow \left|\begin{array}{l} \alpha = \dfrac{y_0 - 2y_1 + y_2}{2(\Delta x)^2} \\ \beta = \dfrac{y_2 - y_0}{2(\Delta x)} \\ \gamma = y_1 \end{array}\right.$$

et $S = \dfrac{\Delta x}{3}(y_0 + 4y_1 + y_2)$.

$$\text{Par suite, il vient : } \left|\begin{array}{lcl} S_1 & = & \dfrac{\Delta x}{3}(y_0 + 4y_1 + y_2) \\ S_2 & = & \dfrac{\Delta x}{3}(y_2 + 4y_3 + y_4) \\ S_3 & = & \dfrac{\Delta x}{3}(y_4 + 4y_5 + y_6) \\ \vdots & = & \\ S_{n/2} & = & \dfrac{\Delta x}{3}(y_{n-2} + 4y_{n-1} + y_n) \end{array}\right.$$

d'où

$$I = \int_a^b f(x)dx \approx \frac{\Delta x}{3}\left(f(a) + 4\sum_{i=1,3,5\ldots}^{n-1} f(x_i) + 2\sum_{i=2,4,6\ldots}^{n-2} f(x_i) + f(b)\right)$$

C'est la formule dite de Simpson qui repose sur une approximation de f par des arcs de parabole.

Ecrire une fonction `simpson_integration` *qui calcule l'intégrale définie I d'une fonction f sur $[a, b]$ à l'ordre n par la méthode de Simpson.*

TD 3.24 : Tracés de courbes paramétrées

Une courbe paramétrée dans le plan est une courbe où l'abscisse x et l'ordonnée y sont des fonctions d'un paramètre qui peut être le temps t ou un angle θ par exemple. La courbe se présente donc sous la forme $x = f(t), y = g(t)$. Les tableaux ci-dessous en donnent quelques exemples.

droite	$x = x_0 + \alpha t$
	$y = y_0 + \beta t$
cercle	$x = x_0 + r\cos(\theta)$
	$y = y_0 + r\sin(\theta)$
ellipse	$x = x_0 + a\cos(\phi)$
	$y = y_0 + b\cos(\phi)$
hyperbole	$x = x_0 + \dfrac{a}{\cos(\theta)}$
	$y = y_0 + b\tan(\theta)$

cycloïde	$x = x_0 + r(\phi - \sin(\phi))$
	$y = y_0 + r(1 - \cos(\phi))$
épicycloïde	$x = x_0 + (R+r)\cos(\theta) - r\cos\left(\dfrac{R+r}{r}\cdot\theta\right)$
	$y = y_0 + (R+r)\sin(\theta) - r\sin\left(\dfrac{R+r}{r}\cdot\theta\right)$
hypercycloïde	$x = x_0 + (R-r)\cos(\theta) + r\cos\left(\dfrac{R-r}{r}\cdot\theta\right)$
	$x = y_0 + (R-r)\sin(\theta) + r\sin\left(\dfrac{R-r}{r}\cdot\theta\right)$

limaçon de Pascal	$x = x_0 + (a\cos(\theta) + b)\cos(\theta)$
	$y = y_0 + (a\cos(\theta) + b)\sin(\theta)$
spirale logarithmique	$x = x_0 + ke^{\theta}\cos(\theta)$
	$y = y_0 + ke^{\theta}\sin(\theta)$

Ecrire une fonction `drawCurve` *qui permettent le tracé de telles courbes paramétrées (figure 3.19). On utilisera les instructions* à la LOGO *pour réaliser ces tracés (voir annexe 3.5.1 page 152).*

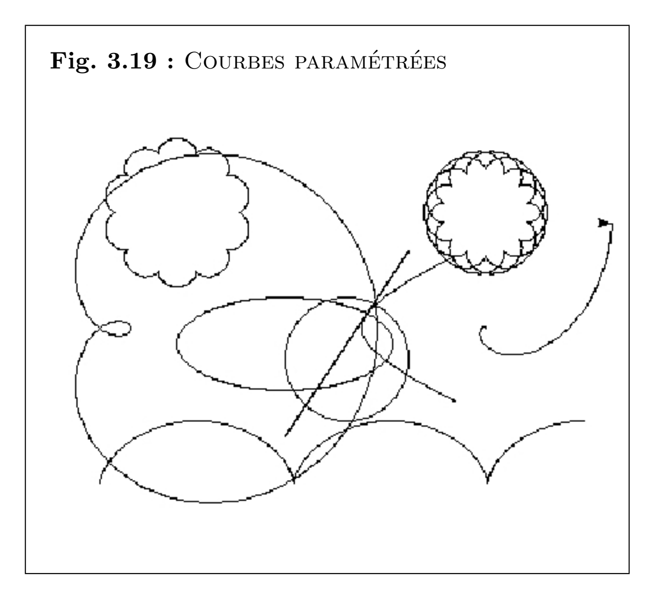


Fig. 3.19 : COURBES PARAMÉTRÉES

3.4.5 Solutions des exercices

TD 3.13 : QCM (3).

Les bonnes réponses sont extraites directement de ce document.

1c, 2a, 3c, 4a, 5c, 6a, 7c, 8a, 9c, 10d, 11d

TD 3.14 : Passage des paramètres.

1. `u = f(t)`

```
x = t
y = x + 2
tmp = y
del x, y
u = tmp
del tmp
```

2. `u = g(t)`

```
z = t
x = z
y = x + 2
tmp = y
del x, y
v = 2*tmp
tmp = v
del v, z
u = tmp
del tmp
```

3. `u = h(t)`

```
a = t
x = a
y = x + 2
tmp = y
del x, y
z = tmp
del tmp
x = z
y = x + 2
tmp = y
del x, y
v = 2*tmp
tmp = v
del v, z
b = tmp
del tmp
tmp = b
del a, b
u = tmp
del tmp
```

TD 3.15 : Portée des variables (2).

1.
```
>>> x = 5
>>> x
5
>>> x = x + 2
>>> x
7
>>> x = 2*(x+2)
>>> x
18
```

2.
```
>>> x = 5
>>> x
5
>>> y = f(x)
f 7
>>> x, y
(5, 7)
>>> z = 2*f(y)
f 9
>>> x, y, z
(5, 7, 18)
```

3.
```
>>> x = 5
>>> x
5
>>> z = 2*f(f(x))
f 7
f 9
>>> x, z
(5, 18)
```

4.
```
>>> x = 5
>>> x
5
>>> f(x)
f 7
7
>>> x
5
>>> g(x)
f 7
g 14
14
>>> x
5
>>> h(x)
f 7
f 9
g 18
h 18
18
>>> x
5
```

TD 3.16 : Suite géométrique.

$$s = \sum_{k=0}^{n} a \cdot b^k = a\frac{b^{n+1}-1}{b-1}$$

```
def sommeGeometrique(a,b,n):
    """
    s = sommeGeometrique(a,b,n)
    est la somme des n premiers termes d'une suite
    géométrique a*b^k (k in [0,n])
    """
    assert type(n) is int and n >= 0
    if n == 0: s = 1
    else: s = sommeGeometrique(a,b,n-1) + (a * b**n)
    return s
```

TD 3.17 : Puissance entière.

$$p = x^n$$

```
def puissance(x,n):
    """
    y = puissance(x,n)
    est la puissance entière de x de degré n
    """
    assert type(n) is int and n >= 0
    if n == 0: p = 1
    else: p = x*puissance(x,n-1)
    return p
```

TD 3.18 : Coefficients du binôme.

$$(a+b)^n = \sum_{k=0}^{n} \frac{n!}{k!(n-k)!} x^{n-k} y^k$$

```
def coefficientBinome(n,p):
    """
    c = coefficientBinome(n,p)
    est le p-ième coefficient du binôme (a+b)**n
    """
    assert type(n) is int and type(p) is int
    assert n >= 0 and p >= 0 and p <= n
    if p == 0 or n == 0 or n == p: c = 1
    else: c = coefficientBinome(n-1,p) + coefficientBinome(n-1,p-1)
    return c
```

TD 3.19 : Fonction d'Ackerman.

$$f : N^2 \rightarrow N \left\{ \begin{array}{lcl} f(0,n) & = & n+1 \\ f(m,0) & = & f(m-1,1) \text{ si } m > 0 \\ f(m,n) & = & f(m-1,f(m,n-1)) \text{ si } m > 0, n > 0 \end{array} \right.$$

```
def ackerman(m,n):
    """
    y = ackerman(m,n)
    """
    assert type(n) is int and type(m) is int
    assert n >= 0 and m >= 0
    if m == 0: a = n + 1
    elif: n == 0: a = ackerman(m-1,1)
    else: a = ackerman(m-1,ackerman(m,n-1))
    return a
```

TD 3.20 : Addition binaire.

```
def add2(code1,code2):
    """
    sum2 = add2(code1,code2)
    additionn binaire sum2 = code1 + code2

    >>> add2([1,0,1],[1])
    [1, 1, 0]
    >>> add2([1,0,1],[1,0])
    [1, 1, 1]
    >>> add2([1,0],[1,0,1])
    [1, 1, 1]
    >>> add2([1,0,1],[1,1])
    [1, 0, 0, 0]
    """
    assert type(code1) is list
    assert type(code2) is list

    sum2 = []
    diffLen = len(code1) - len(code2)
    if diffLen > 0:
         for i in range(diffLen): insert(code2,0,0)
    else:
         for i in range(-diffLen): insert(code1,0,0)

    for i in range(len(code1)): append(sum2,0)

    carry = 0
    for i in range(len(code1)-1,-1,-1):
         value = code1[i] + code2[i] + carry
         if value >= 2:
             sum2[i] = value - 2
             carry = 1
         else:
             sum2[i] = value
             carry = 0

    if carry == 1: insert(sum2,0,1)

    return sum2
```

TD 3.21 : Complément à 2.

```
def neg2(code):
    """
    neg = neg2(code)
    complément à 2 d'un entier binaire

    >>> neg2([0,0,0,1,0,1,1,1])
    [1, 1, 1, 0, 1, 0, 0, 1]
    >>> neg2([1, 1, 1, 0, 1, 0, 0, 1])
    [0, 0, 0, 1, 0, 1, 1, 1]

    >>> for a in [0,1]:
    ...     for b in [0,1]:
    ...         for c in [0,1]:
    ...             add2([a,b,c],neg2([a,b,c]))
    [0, 0, 0]
    [1, 0, 0, 0]
    [1, 0, 0, 0]
    [1, 0, 0, 0]
    [1, 0, 0, 0]
    [1, 0, 0, 0]
    [1, 0, 0, 0]
    [1, 0, 0, 0]
    """

    assert type(code) is list
    neg = []

    carry = 1
    for i in range(len(code)): append(neg,int(not code[i]))
    for i in range(len(code)):
         value = neg[len(code)-1-i] + carry
         if value >= 2:
             neg[len(code)-1-i] = value - 2
             carry = 1
         else:
             neg[len(code)-1-i] = value
             carry = 0

    return neg
```

TD 3.22 : Codage-décodage des réels.

```
def ieee(x):
    """
    ieee_code = ieee(x)
    code le réel x selon la norme IEEE 754 simple précision

    >>> ieee(0.0)
    [0,
     0, 0, 0, 0, 0, 0, 0, 0,
     0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
    >>> ieee(0.625)
    [0,
     0, 1, 1, 1, 1, 1, 1, 0,
     0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
    >>> ieee(3.1399998664855957)
    [0,
     1, 0, 0, 0, 0, 0, 0, 0,
     1, 0, 0, 1, 0, 0, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 0, 0, 0, 0, 1, 0]
    >>> ieee(-4573.5)
    [1,
     1, 0, 0, 0, 1, 0, 1, 1,
     0, 0, 0, 1, 1, 1, 0, 1, 1, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
    """
    assert type(x) is float
    ieee_code = []

    k_exponent = 8
    k_significand = 23
    k_ieee = 32
    bias = code(127,2,k_exponent)

    x_int = int(abs(x))
    x_frac = abs(x) - x_int
    expo_2 = 0

    for i in range(k_ieee): ieee_code.append(0)
```

```
    # calcul du signe
    sign = int(x < 0)

    # calcul de la mantisse
    i = 0
    significand = []
    while (x_int != 0) and (i < k_significand):
         significand.insert(0,x_int%2)
         x_int = x_int/2
         i = i + 1

    if len(significand) > 0 and
       significand[0] == 1:
         del significand[0]
         expo_2 = len(significand)

    i = len(significand)
    while (x_frac != 0) and (i < k_significand):
         x_frac = x_frac * 2
         x_int = int(x_frac)
         x_frac = x_frac - x_int
         if (x_int == 0) and (i == 0):
             expo_2 = expo_2 - 1
         else:
             significand.append(x_int)
             i = i + 1

    if abs(x) < 1 and len(significand) > 0 and
       significand[0] == 1:
         del significand[0]
         expo_2 = expo_2 - 1

    for i in range(len(significand),k_significand):
         significand.append(0)

    # calcul de l'exposant
    exponent = code(abs(expo_2),2,k_exponent)

    if expo_2 >= 0: exponent = add2(bias,exponent)
    elif expo_2 < 0: exponent = sub2(bias,exponent)

    # calcul du code IEEE 754 simple précision
    if x == 0.0:
         ieee_code = []
         for i in range(k_ieee): ieee_code.append(0)
    else:
         ieee_code[0] = sign
         ieee_code[1:9] = exponent
         ieee_code[9:32] = significand

    return ieee_code

#-----------------------------------------------
def sub2(code1,code2):
    """
    substract = sub2(code1,code2)
    soustraction binaire substract = code1 - code2
    """
    assert type(code1) is list
    assert type(code2) is list
    assert len(code1) == len(code2)
    substract = []
    for i in range(len(code1)): append(substract,0)

    carry = 0
    for i in range(len(code1)-1,-1,-1):
         if code1[i] < (code2[i] + carry):
             substract[i] = code1[i] + 2 - (code2[i] + carry)
             carry = 1
         else:
             substract[i] = code1[i] - (code2[i] + carry)
             carry = 0

    return substract
```

TD 3.23 Intégration numérique.

1. Méthode des rectangles.

$$I = \int_a^b f(x)dx \approx (f(x_1) + f(x_2) + \cdots + f(x_n))\,\Delta x$$

Intégration : méthode des rectangles

```python
def rectangle_integration(f,x1,x2,n):
    """
    intégration de f(x) entre x1 et x2
    par la méthode des n rectangles

    >>> fabs(rectangle_integration(sin,0.,2*pi,100000)) < 1.e-6
    True
    >>> fabs(rectangle_integration(sin,0.,pi,100000) - 2.) < 1.e-6
    True
    >>> fabs(rectangle_integration(sin,0.,pi/2,100000) - 1) < 1.e-6
    True
    >>> fabs(rectangle_integration(cos,0.,pi,100000)) < 1.e-6
    True
    >>> fabs(rectangle_integration(cos,-pi/2,pi/2,100000) - 2) < 1.e-6
    True
    """
    assert type(x1) is float
    assert type(x2) is float
    assert x1 <= x2

    integral = 0.0
    width = (x2 - x1)/n
    x = x1 + width/2
    while x < x2:
        integral = integral + f(x)
        x = x + width
    integral = integral*width
    return integral
```

2. Méthode des trapèzes.

$$I = \int_a^b f(x)dx \approx \left(\frac{f(a)+f(b)}{2} + f(x_1) + f(x_2) + \cdots + f(x_{n-1})\right)\Delta x$$

Intégration : méthode des trapèzes

```
def trapezoid_integration(f,x1,x2,n):

    """
    intégration de f(x) entre x1 et x2
    par la méthode des n trapèzes

    >>> fabs(trapezoid_integration(sin,0.,2*pi,100000)) < 1.e-6
    True
    >>> fabs(trapezoid_integration(sin,0.,pi,100000) - 2.) < 1.e-6
    True
    >>> fabs(trapezoid_integration(sin,0.,pi/2,100000) - 1) < 1.e-6
    True
    >>> fabs(trapezoid_integration(cos,0.,pi,100000)) < 1.e-6
    True
    >>> fabs(trapezoid_integration(cos,-pi/2,pi/2,100000) - 2) < 1.e-6
    True
    """
    assert type(n) is int
    assert type(x1) is float
    assert type(x2) is float
    assert x1 <= x2

    integral = (f(x1) + f(x2))/2
    width = (x2 - x1)/n
    x = x1 + width
    while x < x2:
        integral = integral + f(x)
        x = x + width
    integral = integral*width
    return integral
```

3. Méthode de Simpson.

$$I = \int_a^b f(x)dx \approx \frac{\Delta x}{3}\left(f(a) + 4 \cdot \sum_{i=1,3,5\ldots}^{n-1} f(x_i) + 2 \cdot \sum_{i=2,4,6\ldots}^{n-2} f(x_i) + f(b)\right)$$

Intégration : méthode de Simpson

```
def simpson_integration(f,x1,x2,n):
    """
    intégration de f(x) entre x1 et x2
    par la méthode de Simpson

    >>> fabs(simpson_integration(sin,0.,2*pi,100000)) < 1.e-6
    True
    >>> fabs(simpson_integration(sin,0.,pi,100000) - 2.) < 1.e-6
    True
    >>> fabs(simpson_integration(sin,0.,pi/2,100000) - 1) < 1.e-6
    True
    >>> fabs(simpson_integration(cos,0.,pi,100000)) < 1.e-6
    True
    >>> fabs(simpson_integration(cos,-pi/2,pi/2,100000) - 2) < 1.e-6
    True
    """
    assert type(n) is int
    assert type(x1) is float
    assert type(x2) is float
    assert x1 <= x2
    assert n%2 == 0

    integral = f(x1) + f(x2)
    width = (x2 - x1)/n
    for i in range(1,n,2):
        integral = integral + 4*f(x1 + i*width)
    for i in range(2,n,2):
        integral = integral + 2*f(x1 + i*width)
    integral = integral*width/3
    return integral
```

TD 3.24 Tracés de courbes paramétrées.

```
def drawCurve((fx,fy),t1,t2,dt):
    """
    trace une courbe paramétrée pour t dans [t1,t2] par pas de dt
    pour les fonctions x = fx(t) et y = fy(t)

    >>> drawCurve(parametric_line(10,-10,2,3),-20.,20.,0.1)
    >>> drawCurve(parametric_circle(10,-20,40),0.,2*pi,pi/100)
    >>> drawCurve(parametric_ellipse(-30.,-10.,70,30),0.,2*pi,pi/100)
    >>> drawCurve(parametric_hyperbola(-50.,0.,70,30),-1.,1.,0.1)
    >>> drawCurve(parametric_cycloid(-150.,-100.,20.),0.,5*pi,pi/100)
    >>> drawCurve(parametric_epicycloid(-100.,75.,40.,4.),0.,2*pi,pi/100)
    >>> drawCurve(parametric_hypercycloid(100.,75.,40.,6.),0.,8*pi,pi/100)
    >>> drawCurve(pascal_snail(-150.,0.,100.,80.),0.,2*pi,pi/100)
    >>> drawCurve(logarithmic_spiral(100.,0.,0.1),0.,7.,pi/50)
    """
    assert type(t1) is float
    assert type(t2) is float
    assert type(dt) is float

    values = []
    t = t1 + dt
    while t < t2:
        append(values,t)
        t = t + dt
    up()
    goto(fx(t1),fy(t1))
    down()
    for t in values:
        goto(fx(t),fy(t))
    return
```

La fonction `drawCurve` nécessite de passer en argument 2 noms de fonction `fx` et `fy` faisant référence respectivement à l'équation paramétrique $x(t)$ et $y(t)$. Par exemple, pour un cercle de rayon $r = 1$, on pourra appeler la fonction `drawCurve` de la manière suivante : `drawCurve((cos,sin),0.,2*pi,pi/100)`. Si l'on veut tracer, un cercle de rayon $r = 40$ centré en $(x_0 = -10, y_0 = 5)$, on pourra définir les fonctions `cx` et `cy` telles que :

```
def cx(t):
    x0 = -10
    r = 40
    return x0 + r*cos(t)
```

```
def cy(t):
    y0 = 5
    r = 40
    return y0 + r*sin(t)
```

et appeler `drawCurve` de la manière suivante : `drawCurve((cx,cy),0.,2*pi,pi/100)`. Malheureusement, ce dernier procédé n'est pas suffisamment générique. En effet, si on veut maintenant tracer un cercle de rayon $r = 20$ centré en $(0, -34)$, il faudra redéfinir les fonctions `cx` et `cy` ou définir 2 nouvelles fonctions pour ce nouveau tracé. Ainsi, dès que l'on voudra dessiner un cercle de rayon r centré en (x_0, y_0), il faudra recommencer cette opération : ce qui n'est finalement pas très opérationnel. Pour être plus efficace, on utilisera la directive `lambda` du langage Python qui permet de définir une fonction anonyme (sans nom) qui peut être retournée par une autre fonction. Ainsi, pour le cercle, on définira la fonction `parametric_circle` de façon à ce qu'elle retourne un doublet de fonctions anonymes paramétrables :

```
def parametric_circle(x0,y0,r):
    return lambda(t): x0 + r * cos(t), lambda(t): y0 + r * sin(t)
```

Nous pourrons alors appeler la fonction `drawCurve` avec comme premier argument un appel à la fonction `parametric_circle` qui sera remplacée par le doublet de fonctions anonymes :

- pour un cercle de rayon $r = 40$ centré en $(10, 5)$:

```
drawCurve(parametric_circle(10,5,40),0.,2*pi,pi/100)
```

- pour un cercle de rayon $r = 20$ centré en $(0, -34)$:

```
drawCurve(parametric_circle(0,20,-34),0.,2*pi,pi/100)
```

- et de manière générale pour un cercle de rayon r centré en (x_0, y_0) :

```
drawCurve(parametric_circle(x0,y0,r),0.,2*pi,pi/100)
```

Nous donnons ci-dessous 9 exemples de fonctions pour définir des familles de courbes paramétrées : droites, cercles, ellipses, hyperboles, cycloïdes, épicycloïdes, hypocycloïdes, limaçons de Pascal et spirales algorithmiques.

Droite paramétrique

```
def parametric_line(x0,y0,alpha,beta):
    """
    droite paramétrique
    x = x0 + alpha*t, y = y0 + beta*t
    """
    return lambda(t): x0 + alpha*t,
           lambda(t): y0 + beta*t
```

Cercle paramétrique

```
def parametric_circle(x0,y0,r):
    """
    cercle paramétrique
    x = x0 + r*cos(theta), y = y0 + r * sin(theta)
    """
    return lambda(theta): x0 + r * cos(theta),
           lambda(theta): y0 + r * sin(theta)
```

Ellipse paramétrique

```
def parametric_ellipse(x0,y0,a,b):
    """
    ellipse paramétrique
    x = x0 + a*cos(phi), y = y0 + b*sin(phi)
    """
    return lambda(phi): x0 + a*cos(phi),
           lambda(phi): y0 + b*sin(phi)
```

Hyperbole paramétrique

```
def parametric_hyperbola(x0,y0,a,b):
    """
    hyperbole paramétrique
    x = x0 + a/cos(theta), y = y0 + b*tan(theta)
    """
    return lambda(theta): x0 + a/cos(theta),
           lambda(theta): y0 + b*tan(theta)
```

Cycloïde paramétrique

```
def parametric_cycloid(x0,y0,r):
    """
    cycloïde paramétrique
    x = x0 + r*(phi-sin(phi)), y = y0 + r*(1-cos(phi))
    """
    return lambda(phi): x0 + r*(phi-sin(phi)),
           lambda(phi): y0 + r*(1-cos(phi))
```

Epicycloïde paramétrique

```
def parametric_epicycloid(x0,y0,R,r):
    """
    épicycloïde paramétrique
    x = x0 + (R+r)*cos(theta) - r*cos(theta*(R+r)/r),
    x = y0 + (R+r)*sin(theta) - r*sin(theta*(R+r)/r)
    """
    return lambda(theta): x0 + (R+r)*cos(theta) - r*cos(theta*(R+r)/r),
           lambda(theta): y0 + (R+r)*sin(theta) - r*sin(theta*(R+r)/r)
```

Hypercycloïde paramétrique

```
def parametric_hypercycloid(x0,y0,R,r):
    """
    hypercycloïde paramétrique
    x = x0 + (R-r)*cos(theta) + r*cos(theta*(R-r)/r),
    x = y0 + (R-r)*sin(theta) + r*sin(theta*(R-r)/r)
    """
    return lambda(theta): x0 + (R-r)*cos(theta) + r*cos(theta*(R-r)/r),
           lambda(theta): y0 + (R-r)*sin(theta) + r*sin(theta*(R-r)/r)
```

Limaçon de Pascal

```
def pascal_snail(x0,y0,a,b):
    """
    limaçon de Pascal
    x = x0 + (a*cos(theta) + b)*cos(theta)
    y = y0 + (a*cos(theta) + b)*sin(theta)
    """
    return lambda(theta): x0 + (a*cos(theta) + b)*cos(theta),
           lambda(theta): y0 + (a*cos(theta) + b)*sin(theta)
```

Spirale logarithmique

```
def logarithmic_spiral(x0,y0,k):
    """
    spirale logarithmique
    x = x0 + k*exp(theta)*cos(theta)
    y = y0 + k*exp(theta)*sin(theta)
    """
    return lambda(theta): x0 + k*exp(theta)*cos(theta),
           lambda(theta): y0 + k*exp(theta)*sin(theta)
```

Remarque : Ce procédé qui consiste à passer une fonction `lambda` en argument d'une fonction aurait pu être utilisé dans le cas des fonctions d'intégration numérique du TD 3.23 précédent.

Exemples :

```
>>> f = lambda(x): 3*x + 5
>>> simpson_integration(f,-1.,1.,100)
10.0
>>> g = lambda(x): sin(cos(x))
>>> simpson_integration(g,-pi/2.,pi/2.,10000)
1.7864874819500629
```

3.5 Annexes

3.5.1 Instructions Logo

> Logo is the name for a philosophy of education and a continually evolving family of programming languages that aid in its realization (Harold Abelson, Apple Logo, 1982). *This statement sums up two fundamental aspects of Logo and puts them in the proper order. The Logo programming environments that have been developed over the past 28 years are rooted in constructivist educational philosophy, and are designed to support constructive learning. [...] Constructivism views knowledge as being created by learners in their own minds through interaction with other people and the world around them. This theory is most closely associated with Jean Piaget, the Swiss psychologist, who spent decades studying and documenting the learning processes of young children.*
>
> **Logo Foundation :** http://el.media.mit.edu/logo-foundation

On suppose connues les procédures de tracés géométriques *à la* Logo :

`degrees()` fixe l'unité d'angle en degrés

`radians()` fixe l'unité d'angle en radians

`reset()` efface l'écran et réinitialise les variables

`clear()` efface l'écran

`up()` lève le crayon

`down()` abaisse le crayon

`forward(d)` avance d'une distance d

`backward(d)` recule d'une distance d

`left(a)` tourne sur la gauche d'un angle a

`right(a)` tourne sur la droite d'un angle a

`goto(x,y)` déplace le crayon à la position (x, y)

`towards(x,y)` oriente vers le point de coordonnées (x, y)

`setheading(a)` oriente d'un angle a par rapport à l'axe des x

`position()` donne la position (x, y) du crayon

`heading()` donne l'orientation a du déplacement

`circle(r)` trace un cercle de rayon r

`circle(r,a)` trace un arc de cercle de rayon r et d'angle au sommet a.

3.5.2 Fonctions PYTHON prédéfinies

Les principales fonctions prédéfinies en PYTHON sont listées dans les tableaux ci-dessous, extraits du PYTHON *2.5 Quick Reference Guide* [10].

Function	**Result**
`abs(x)`	Returns the absolute value of the number `x`.
`all(iterable)`	Returns `True` if `bool(x)` is `True` for all values `x` in the `iterable`.
`any(iterable)`	Returns `True` if `bool(x)` is `True` for any values `x` in the `iterable`.
`bool([x])`	Converts a value to a Boolean, using the standard truth testing procedure. If `x` is false or omitted, returns `False` ; otherwise returns `True`.
`chr(i)`	Returns one-character string whose ASCII code is integer `i`.
`cmp(x,y)`	Returns negative, 0, positive if `x` `<`, `==`, `>` to `y` respectively.
`complex(real[, image])`	Creates a complex object (can also be done using `J` or `j` suffix, e.g. `1+3J`).
`dict([mapping-or-sequence])`	Returns a new dictionary initialized from the optional argument (or an empty dictionary if no argument). Argument may be a sequence (or anything iterable) of pairs (key,value).
`dir([object])`	Without args, returns the list of names in the current local symbol table. With a module, class or class instance `object` as arg, returns the list of names in its attr. dictionary.
`divmod(a,b)`	Returns tuple (`a//b`, `a%b`).
`enumerate(iterable)`	Iterator returning pairs (index, value) of `iterable`, e.g. `List(enumerate('Py'))` → `[(0, 'P'), (1, 'y')]`.
`eval(s[, globals[, locals]])`	Evaluates string `s`, representing a single python expression, in (optional) `globals`, `locals` contexts. Example : `x = 1 ; assert eval('x + 1') == 2`
`execfile(file[, globals[,locals]])`	Executes a `file` without creating a new module, unlike import.
`filter(function,sequence)`	Constructs a list from those elements of `sequence` for which `function` returns true. `function` takes one parameter.
`float(x)`	Converts a number or a string to floating point.
`globals()`	Returns a dictionary containing the current global variables.
`help([object])`	Invokes the built-in help system. No argument → interactive help ; if `object` is a string (name of a module, function, class, method, keyword, or documentation topic), a help page is printed on the console ; otherwise a help page on `object` is generated.
`hex(x)`	Converts a number `x` to a hexadecimal string.
`id(object)`	Returns a unique integer identifier for `object`.

Function	**Result**
`input([prompt])`	Prints `prompt` if given. Reads input and evaluates it.
`int(x[, base])`	Converts a number or a string to a plain integer. Optional `base` parameter specifies base from which to convert string values.
`len(obj)`	Returns the length (the number of items) of an object (sequence, dictionary).
`list([seq])`	Creates an empty list or a list with same elements as `seq`. `seq` may be a sequence, a container that supports iteration, or an iterator object. If `seq` is already a list, returns a copy of it.
`locals()`	Returns a dictionary containing current local variables.
`map(function, sequence)`	Returns a list of the results of applying `function` to each item from `sequence`(s).
`oct(x)`	Converts a number to an octal string.
`open(filename[,mode='r',[bufsize]])`	Returns a new file object. `filename` is the file name to be opened. `mode` indicates how the file is to be opened (`'r', 'w', 'a', '+', 'b', 'U'`). `bufsize` is 0 for unbuffered, 1 for line buffered, negative or omitted for system default, `>1` for a buffer of (about) the given size.
`ord(c)`	Returns integer ASCII value of `c` (a string of len 1).
`range([start,] end [, step])`	Returns list of ints from `>= start` and `< end`. With 1 arg, list from `0..arg-1`. With 2 args, list from `start..end-1`. With 3 args, list from `start` up to `end` by `step`.
`raw_input([prompt])`	Prints `prompt` if given, then reads string from std input (no trailing `n`).
`reload(module)`	Re-parses and re-initializes an already imported `module`.
`repr(object)`	Returns a string containing a printable and if possible evaluable representation of an `object`. ≡ `` `object` `` (using backquotes).
`round(x, n=0)`	Returns the floating point value `x` rounded to `n` digits after the decimal point.
`str(object)`	Returns a string containing a nicely printable representation of an `object`.
`sum(iterable[, start=0])`	Returns the sum of a sequence of numbers (not strings), plus the value of parameter. Returns `start` when the sequence is empty.
`tuple([seq])`	Creates an empty tuple or a tuple with same elements as `seq`.
`type(obj)`	Returns a type object representing the type of `obj`.
`xrange(start [, end [, step]])`	Like `range()`, but doesn't actually store entire list all at once. Good to use in `for` loops when there is a big range and little memory.

3.5.3 Fonctions en Python

Le principe de la définition d'une fonction en Python est présentée ci-dessous (d'après le Python *2.5 Quick Reference Guide* [10]).

`def funcName([paramList]) :`
`    block`
	Creates a function object and binds it to name `funcName`.
	`paramList : := [param [, param]*]`
	`param : := value | id=value | *id | **id`

- Arguments are passed by value, so only arguments representing a mutable object can be modified (are inout parameters).
- Use `return` to return `None` from the function, or `return value` to return a value. Use a tuple to return more than one value, e.g. `return 1,2,3`.
- Keyword arguments `arg=value` specify a default value (evaluated at function definition time). They can only appear last in the param list, e.g. `foo(x, y=1, s='')`.
- Pseudo-arg `*args` captures a tuple of all remaining non-keyword args passed to the function, e.g. if `def foo(x, *args) : ...` is called `foo(1, 2, 3)`, then `args` will contain `(2,3)`.
- Pseudo-arg `**kwargs` captures a dictionary of all extra keyword arguments, e.g. if `def foo(x, **kwargs) : ...` is called `foo(1, y=2, z=3)`, then `kwargs` will contain {`'y' :2, 'z' :3`}. if `def foo(x, *args, **kwargs) : ...` is called `foo(1, 2, 3, y=4, z=5)`, then `args` will contain `(2, 3)`, and `kwargs` will contain {`'y' :4, 'z' :5`}.
- `args` and `kwargs` are conventional names, but other names may be used as well.
- `*args` and `**kwargs` can be "forwarded" (individually or together) to another function, e.g. `def f1(x, *args, **kwargs) : f2(*args, **kwargs)`.

3.5.4 L'utilitaire pydoc

Cette annexe est un extrait du site officiel PYTHON concernant `pydoc` : http://docs.python.org/lib/module-pydoc.html. La figure 3.20 ci-contre illustre son utilisation.

Fig. 3.20 : DOCUMENTATION EN PYTHON
`pydoc` *est testé ici avec l'exemple qui nous a servi de fil rouge tout au long de la section 3.2.*

```
$ pydoc fibo
Help on module fibo :

NAME
    fibo
FILE
    /home/info/S1/cours/fonctions/fibo.py

FUNCTIONS
    fibonacci(n)
        u = fibonacci(n)
        est le nombre de Fibonacci
        à l'ordre n si n :int >= 0
        >>> fibonacci(0)
        1
        >>> fibonacci(2)
        2
        >>> fibonacci(9)
        55
```

pydoc – Documentation generator and online help system

The pydoc module automatically generates documentation from Python modules. The documentation can be presented as pages of text on the console, served to a Web browser, or saved to HTML files.

The built-in function `help()` invokes the online help system in the interactive interpreter, which uses `pydoc` to generate its documentation as text on the console. The same text documentation can also be viewed from outside the PYTHON interpreter by running `pydoc` as a script at the operating system's command prompt. For example, running
`pydoc sys`
at a shell prompt will display documentation on the `sys` module, in a style similar to the manual pages shown by the Unix `man` command. The argument to `pydoc` can be the name of a function, module, or package, or a dotted reference to a class, method, or function within a module or module in a package. If the argument to `pydoc` looks like a path (that is, it contains the path separator for your operating system, such as a slash in Unix), and refers to an existing PYTHON source file, then documentation is produced for that file.

Specifying a `-w` flag before the argument will cause HTML documentation to be written out to a file in the current directory, instead of displaying text on the console.

Specifying a `-k` flag before the argument will search the synopsis lines of all available modules for the keyword given as the argument, again in a manner similar to the Unix `man` command. The synopsis line of a module is the first line of its documentation string.

You can also use `pydoc` to start an HTTP server on the local machine that will serve documentation to visiting Web browsers. `pydoc -p 1234` will start a HTTP server on port `1234`, allowing you to browse the documentation at `http ://localhost :1234/` in your preferred Web browser. `pydoc -g` will start the server and additionally bring up a small `Tkinter`-based graphical interface to help you search for documentation pages.

When `pydoc` generates documentation, it uses the current environment and path to locate modules. Thus, invoking `pydoc` spam documents precisely the version of the module you would get if you started the PYTHON interpreter and typed "import spam".

Chapitre 4

Structures linéaires



Sommaire

4.1 Introduction

Un algorithme est une suite ordonnée d'instructions qui indique la démarche à suivre pour résoudre une série de problèmes équivalents. Les deux chapitres précédents concernaient la structuration des algorithmes, soit en contrôlant le flux d'instructions à l'aide d'instructions de base appropriées (chapitre 2), soit en factorisant des séquences d'instructions au sein de fonctions et procédures (chapitre 3). Par contre, ils ne s'intéressaient pas explicitement aux données manipulées par ces algorithmes. Ce sera l'objet de ce chapitre d'aborder la structuration des données manipulées par les algorithmes.

Exemple 4.1 : Distance entre 2 points du plan

Soient deux points du plan M_1 et M_2 respectivement de coordonnées (x_1, y_1) et (x_2, y_2). La distance d entre ces 2 points est classiquement calculée par $d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$ dont on déduit l'implémentation informatique suivante :

```
def distance(x1,y1,x2,y2) :
    return sqrt((x2-x1)**2 + (y2-y1)**2)
```

```
>>> distance(0,0,1,1)
1.4142135623730951
```

La fonction `distance` précédente prend 4 arguments (les 4 coordonnées x_1, y_1, x_2, y_2) là où l'utilisateur de la fonction aurait certainement préféré parler plus directement des 2 points M_1 et M_2. Pour cela, il faut être capable de réunir 2 par 2 les coordonnées des 2 points au sein d'une même collection de données (figure 4.1) et passer des points plutôt que des coordonnées. C'est ce qui est proposé dans l'implémentation suivante en utilisant les n-uplets de Python où la fonction ne prend plus que 2 arguments (les 2 points M_1 et M_2).

```
def distance(m1,m2) :
    return sqrt((m2[0]-m1[0])**2 + (m2[1]-m1[1])**2)
```

```
>>> m1, m2 = (0,0), (1,1)
>>> distance(m1,m2)
1.4142135623730951
```

■ TD4.1

Ce regroupement de coordonnées permet de définir la notion de point là où ne manipulait initialement que des coordonnées. On introduit ainsi de nouveaux types de données.

4.1.1 Types de données

En informatique, une donnée est la représentation d'une information sous une forme conventionnelle adaptée à son exploitation. Une variable est alors un objet informatique qui associe un nom à cette représentation (section 2.2.1 page 42) et qui, selon les langages, est implicitement ou explicitement typée. On distingue classiquement les types de base comme les booléens

Fig. 4.1 : Définitions de l'Académie (9)

DONNÉE *n. f. XIIIe siècle, au sens de « distribution, aumône » ; XVIIIe siècle, comme terme de mathématiques. Participe passé féminin substantivé de donner au sens de « indiquer, dire ». 1. Fait ou principe indiscuté, ou considéré comme tel, sur lequel se fonde un raisonnement ; constatation servant de base à un examen, une recherche, une découverte.* MATH. *Chacune des quantités ou propriétés mentionnées dans l'énoncé d'un problème et qui permettent de le résoudre.* INFORM. *Représentation d'une information sous une forme conventionnelle adaptée à son exploitation.*

COLLECTION *n. f. XIVe siècle, au sens de « amas de pus » ; XVIIe siècle, au sens moderne. Emprunté du latin collectio, « action de recueillir, de rassembler », « ce qui est recueilli ». 1. Ensemble d'objets de même sorte que l'on réunit volontairement dans un esprit de curiosité, ou pour leur valeur artistique, scientifique ou documentaire.*

TD 4.1 : Distance de 2 points de l'espace

Définir la fonction `distance` *qui calcule la distance entre 2 points M_1 et M_2 de l'espace, respectivement de coordonnées (x_1, y_1, z_1) et (x_2, y_2, z_2).*

(type `bool`, {`False`,`True`}), les « entiers » (type `int`, en fait un sous-ensemble des nombres relatifs $\mathbb{Z}$) et les « réels » (type `float`, en fait un sous-ensemble des nombres rationnels $\mathbb{Q}$) des types structurés qui sont des regroupements de types tels que les listes (type `list`, exemple : `[0,3.14,[4,True]]`), les n-uplets (type `tuple`, exemple : `(0,3.14,[4,(True,8)])` ou les dictionnaires (type `dict`, exemple : {`'a' : [1,2,3], 'b' : 5`}).

Définition 4.1 : type de données
Le type d'une variable définit l'ensemble des valeurs qu'elle peut prendre et l'ensemble des opérations qu'elle peut subir.

Remarque 4.1 : *En Python, les principaux types de base sont les booléens (`bool`), les entiers (`int`), les réels (`float`), les complexes (`complex`), les chaînes de caractères (`str`), les n-uplets (`tuple`), les listes (`list`), les ensembles (`set`), les fichiers (`file`) et les dictionnaires (`dict`).*

D'une manière plus formelle, on définit des types abstraits de données (Tad) où les données sont considérées de manière abstraite indépendamment d'une implémentation dans un langage donné sur une machine particulière. On choisit alors une notation pour les décrire ainsi que pour décrire l'ensemble des opérations qu'on peut leur appliquer et les propriétés de ces opérations. Il existe plusieurs manières de définir un type abstrait de données qui diffèrent essentiellement dans la façon de décrire les propriétés des opérations du type. L'annexe 4.6.1 page 193 propose un exemple de type abstrait de séquence fondé sur un formalisme logique [13], mais dans ce qui suit (section 4.2) nous aborderons les séquences de manière plus pragmatique et opérationnelle.

Remarque 4.2 : *On pourra également consulter [9] pour approfondir cette notion de type abstrait de données, en particulier dans le cas des séquences et des principaux algorithmes associés de recherche et de tri.*

On peut ainsi regrouper n'importe quelles données entre elles pour former un nouveau type de données comme :
- un nombre rationnel formé de 2 entiers : le numérateur et le dénominateur (exemple : `(n,d)` pourra représenter le rationnel n/d),
- un nombre complexe formé de 2 réels : la partie réelle et la partie imaginaire (exemple : `(x,y)` pourra représenter le complexe $x + iy$),
- un vecteur de l'espace formé de 3 réels : l'abscisse, l'ordonnée et la côte (exemple : `(x,y,z)` pourra représenter le vecteur de $\mathbb{R}^3$ de composantes (x, y, z)),
- une fiche d'état civil formé de 3 chaînes de caractères et d'un entier : le nom, le prénom, la nationalité et l'âge (exemple : `(nom,prenom,pays,age)`) pourra représenter l'individu `nom prenom` originaire de `pays` et âgé de `age` ans.

Dans les exemples précédents, le nombre d'éléments qui composent le regroupement est connu à l'avance (un rationnel sera toujours composé de 2 entiers, un vecteur de $\mathbb{R}^3$ aura toujours 3 composantes réelles...). Mais ce n'est pas toujours le cas ; on parlera alors de collections de données.

4.1.2 Collections

Définition 4.2 : COLLECTION DE DONNÉES
Une collection est un regroupement fini de données dont le nombre n'est pas fixé a priori.

Ainsi les collections que l'on utilise en informatique sont des objets dynamiques. Le nombre de leurs éléments varie au cours de l'exécution du programme, puisqu'on peut y ajouter et supprimer des éléments en cours de traitement. Plus précisément les principales opérations que l'on s'autorise sur les collections sont les suivantes :
- déterminer le nombre d'éléments de la collection,
- tester l'appartenance d'un élément à la collection,
- ajouter un élément à la collection,
- supprimer un élément de la collection.



Fig. 4.2 : EXEMPLE DE COLLECTION
Tas de chaussures



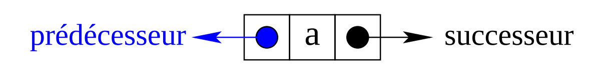
Exemple 4.2 : TAS DE CHAUSSURES
Dans le tas de chaussures de la figure 4.2 ci-contre, il est très facile d'ajouter une paire de chaussures sur le tas : il suffit de la jeter sans précaution sur les autres chaussures. Par contre, pour supprimer une chaussure particulière du tas, ce sera beaucoup plus difficile car il faudra d'abord la retrouver dans cet amas non structuré de chaussures.

A l'inverse du tas de chaussures, les collections informatiques seront structurées pour faciliter et optimiser la recherche d'un élément en leur sein. Les éléments peuvent être de différents types, mais les opérations que l'on effectue sur les collections doivent être indépendantes des types de données des éléments.

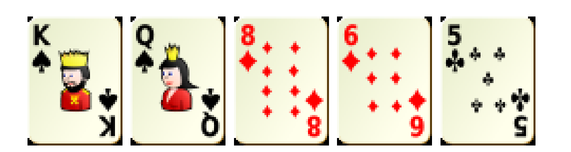
On distingue classiquement 3 grands types de collections : les séquences, les arbres et les graphes.

Définition 4.3 : SÉQUENCE
Une séquence est une suite ordonnée d'éléments, éventuellement vide, accessibles par leur rang dans la séquence.

Fig. 4.3 : ELÉMENTS D'UNE SÉQUENCE



Exemple de séquence : main au poker



Dans une séquence (figure 4.3), chaque élément a un prédecesseur (sauf le premier élément qui n'a pas de prédécesseur) et un successeur (sauf le dernier élément qui n'a pas de successeur). Une liste de noms, une main d'un jeu de cartes, une pile d'assiettes ou une file de spectateurs sont des exemples de structures séquentielles de la vie courante. En PYTHON, on utilisera 3 types de séquences : les chaînes de caractères (type `str`, exemples : `''`, `'bonjour'`, `"ça va ?"`),

les n-uplets (type `tuple`, exemples : `()`, `(1,2,3)`, `('a',2,(1,2,3))`) et les listes (type `list`, exemples : `[]`, `[a,b,c,d,e]`, `[1,'e',(1,2,[x,y])]`).

Définition 4.4 : ARBRE
Un arbre est une collection d'éléments, appelés « nœuds », organisés de façon hiérarchique à partir d'un nœud particulier, appelé la « racine » de l'arbre.

Dans un arbre (figure 4.4), chaque élément a un seul prédecesseur (sauf la racine de l'arbre qui n'a pas de prédecesseur) et peut avoir plusieurs successeurs (on appelle « feuille » de l'arbre un nœud qui n'a pas de successeurs). L'arbre est dit « n-aire » si chaque nœud a au plus n successeurs ; si $n = 2$, on parle d'arbre binaire. Les répertoires d'un système d'exploitation en informatique, la classification des espèces animales en biologie ou le tableau final d'un tournoi de tennis constituent des exemples connus de structures arborescentes.

Exemple 4.3 : TABLEAU FINAL D'UN TOURNOI DE FOOTBALL
Dans l'exemple de la figure 4.5 ci-contre, les nœuds de l'arbre sont les rencontres (exemples : le quart de finale Portugal/Turquie, la demi-finale Pays-Bas/Italie). Il s'agit d'un arbre binaire dont la racine est la finale France/Italie. Les quarts de finale sont les feuilles de cet arbre binaire.

En PYTHON, les ensembles (type `set`) et les dictionnaires (type `dict`) sont implémentés sous forme d'arbres.

Définition 4.5 : GRAPHE
Un graphe est une collection d'éléments, appelés « sommets », et de relations entre ces sommets.

Dans un graphe (figure 4.6), chaque élément peut avoir plusieurs prédécesseurs et plusieurs successeurs. Un même élément peut être à la fois prédécesseur et successeur d'un autre sommet (y compris de lui-même). On parle de « graphe orienté » lorsque les relations entre sommets sont des paires ordonnées de sommets (les relations sont alors appelées « arcs » du graphe) ; on parle de « graphe non orienté » si ce sont des paires non orientées (les relations sont alors appelées « arêtes » du graphe). Un graphe est facilement représentable par un schéma (figure 4.7) où les sommets sont des points et les arcs, des flèches entre deux points (ou les arêtes, des traits entre deux points). Un circuit électronique, une toile d'araignée ou internet sont des exemples de tels graphes.

Les graphes permettent de manipuler plus facilement des objets et leurs relations. L'ensemble des techniques et outils mathématiques mis au point en « Théorie des Graphes » permettent

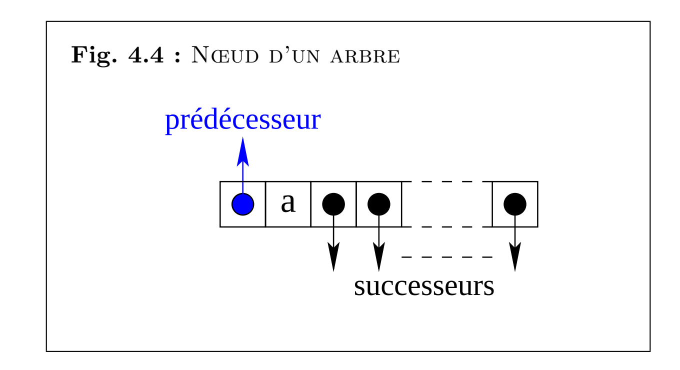


Fig. 4.4 : NŒUD D'UN ARBRE



Fig. 4.5 : EXEMPLE D'ARBRE
Tableau final de l'Euro 2000 de football

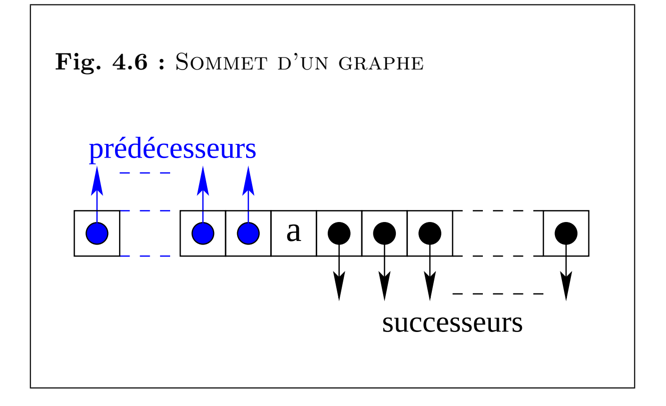


Fig. 4.6 : SOMMET D'UN GRAPHE

de démontrer certaines propriétés des graphes, d'en déduire des méthodes de résolution, des algorithmes...

Exemple 4.4 : Carte routière
Dans la carte routière de la figure 4.8, les villes sont les sommets d'un graphe non orienté dont les arêtes sont les routes qui mènent d'une ville à l'autre. Sur un tel graphe, on peut se poser des questions telles que :
- *Quel est le plus court chemin (en distance ou en temps) pour se rendre d'une ville à une autre ?*
- *Comment optimiser le circuit d'un voyageur de commerce de telle manière qu'il passe une et une seule fois par toutes les villes de son secteur ?*

Fig. 4.7 : Schéma d'un graphe orienté

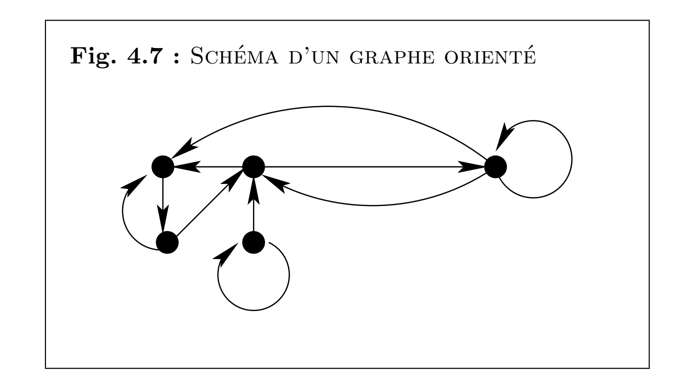


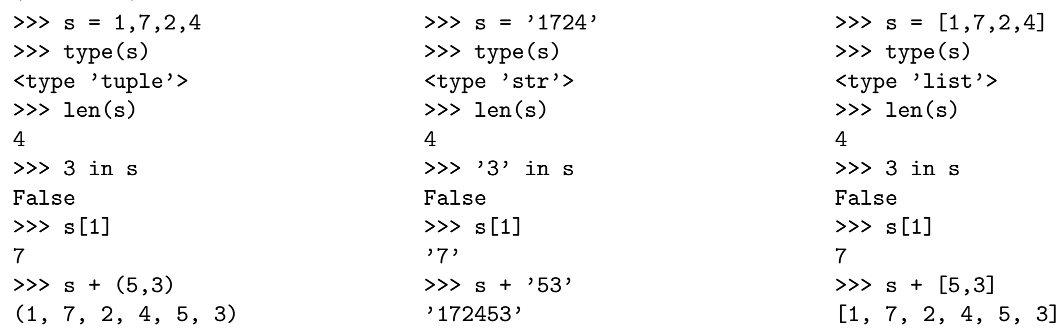
Fig. 4.8 : Exemple de graphe
Carte routière du Finistère Nord



Dans ce chapitre, nous ne nous intéresserons qu'aux structures linéaires : les séquences. Les structures arborescentes et les graphes seront abordés à partir du semestre 4 dans les cours d'informatique de l'ENIB.

4.2 Séquences

Comme en Python, nous distinguerons ici les n-uplets (type `tuple`), les chaînes de caractères (type `str`) et les listes (type `list`) comme autant de variantes de séquences. Quelle que soit la variante considérée, il sera toujours possible de déterminer le type (`type(s)`) et la longueur (`len(s)`) d'une séquence `s`, de tester l'appartenance d'un élément `x` à la séquence `s` (`x in s`), d'accéder à un élément par son rang `i` dans la séquence (`s[i]`) et de concaténer 2 séquences `s1` et `s2` (`s1 + s2`).



```
>>> s = 1,7,2,4
>>> type(s)
<type 'tuple'>
>>> len(s)
4
>>> 3 in s
False
>>> s[1]
7
>>> s + (5,3)
(1, 7, 2, 4, 5, 3)
```

```
>>> s = '1724'
>>> type(s)
<type 'str'>
>>> len(s)
4
>>> '3' in s
False
>>> s[1]
'7'
>>> s + '53'
'172453'
```

```
>>> s = [1,7,2,4]
>>> type(s)
<type 'list'>
>>> len(s)
4
>>> 3 in s
False
>>> s[1]
7
>>> s + [5,3]
[1, 7, 2, 4, 5, 3]
```

Les variantes `tuple`, `str` et `list` diffèrent par leur syntaxe et par le fait qu'une liste est

modifiable alors que les n-uplets et les chaînes de caractères ne le sont pas, au sens où il n'est pas possible de modifier un élément individuel d'une chaîne ou d'un n-uplet.

```
>>> s = 1,7,2,4
>>> type(s)
<type 'tuple'>
>>> s[1] = 3
Traceback ...
TypeError : 'tuple'
object does not support
item assignment
>>> s[1]
7
```

```
>>> s = '1724'
>>> type(s)
<type 'str'>
>>> s[1] = '3'
Traceback ...
TypeError : 'str' object
does not support item
assignment
>>> s[1]
7
```

```
>>> s = [1,7,2,4]
>>> type(s)
<type 'list'>
>>> s[1] = 3
>>> s
[1, 3, 2, 4]
>>> s[1]
3
```

Remarque 4.3 : *Attention ! Par convention, les indices des éléments dans une séquence commencent à* `0`*. Le premier élément a pour indice* `0` *(*`s[0]`*), le deuxième l'indice* `1` *(*`s[1]`*), le troisième l'indice* `2` *(*`s[2]`*) et ainsi de suite jusqu'au dernier qui a l'indice* `n-1` *(*`s[n-1]`*) si* `n` *est le nombre d'éléments dans la séquence (*`n == len(s)`*).*

Remarque 4.4 : *La section 4.6.3 page 194 présente les principales opérations sur les séquences en* PYTHON.

4.2.1 N-uplets

Définition 4.6 : N-UPLET
Un n-uplets est une séquence non modifiable d'éléments.

On parle de singleton quand $n = 1$, de paire quand $n = 2$, de triplet pour $n = 3$, de quadruplet pour $n = 4$... et plus généralement de n-uplet.

D'un point de vue syntaxique, un n-uplet est une suite d'éléments séparés par des virgules. Bien que cela ne soit pas nécessaire, il est conseillé de mettre un n-uplet en évidence en l'enfermant dans une paire de parenthèses, comme PYTHON le fait lui-même. Par ailleurs, il faut toujours au moins une virgule pour définir un n-uplet, sauf pour le n-uplet vide `()`.

```
>>> s = ()
>>> type(s)
<type 'tuple'>
>>> s = 1,7,2,4
>>> type(s)
<type 'tuple'>
>>> s
(1, 7, 2, 4)
```

```
>>> s = (5)
>>> type(s)
<type 'int'>
>>> s = (5,)
>>> type(s)
<type 'tuple'>
>>> s = (5,)+()+(6,7,9)
>>> s
(5, 6, 7, 9)
```

```
>>> s = (5,6,7,9)
>>> s[1:3]
(6, 7)
>>> s[1:]
(6, 7, 9)
>>> s[:2]
(5, 6)
>>> s[-2:]
(7, 9)
```

■TD4.2

TD 4.2 : OPÉRATIONS SUR LES N-UPLETS
Donner un exemple d'utilisation de chacune des opérations sur les n-uplets décrites dans le tableau de la page 195.

TD 4.3 : PGCD ET PPCM DE 2 ENTIERS (2)
voir TD 3.10 Définir une fonction qui calcule le pgcd et le ppcm de 2 entiers a et b.

Les n-uplets sont souvent utilisés pour regrouper sous une même variable plusieurs variables logiquement reliées entre elles comme dans l'exemple des points du plan de l'exemple 4.1 page 158. Ils servent également à retourner plusieurs objets dans une fonction comme dans

les exemples de la fonction standard `divmod` de la section 3.5.2 page 153 ou encore des définitions de courbes paramétriques du TD 3.24 page 136. ■TD4.3

4.2.2 Chaînes de caractères

Définition 4.7 : CHAÎNE DE CARACTÈRES
Une chaîne de caractères est une séquence non modifiable de caractères.

D'un point de vue syntaxique, une chaîne de caractères est une suite quelconque de caractères délimitée soit par des apostrophes (simple *quotes* : `'...'`), soit par des guillemets (double *quotes* : `"..."`). On peut ainsi utiliser des guillemets pour délimiter une chaîne dans laquelle il y a des apostrophes (exemple : `"c'est ça !"`), ou utiliser des apostrophes pour délimiter une chaîne qui contient des guillemets (exemple : `'"hello" dit-il.'`). Pour sa part, le caractère *antislash* (\) permet d'écrire sur plusieurs lignes une chaîne qui serait trop longue pour tenir sur une seule ou bien d'insérer, à l'intérieur d'une chaîne, un certain nombre de caractères spéciaux comme les sauts à la ligne (`\n`), les apostrophes (`\'`), les guillemets (`\"`)... En PYTHON, pour insérer plus aisément des caractères spéciaux dans une chaîne, sans faire usage de l'*antislash*, ou pour faire accepter l'*antislash* lui-même dans la chaîne, on peut délimiter la chaîne à l'aide de triples apostrophes (`'''...'''`) ou de triples guillemets (`"""..."""`).

```
>>> s = 'chaîne entrée sur \
... plusieurs lignes'
>>> s
'chaîne entrée sur plusieurs lignes'
>>> s = 'chaîne entrée \n sur 1 ligne'
>>> print(s)
chaîne entrée
 sur 1 ligne
```

```
>>> s = 'c\'est ça \"peuchère\"'
>>> s
'c\'est ça "peuchère"'
>>> print(s)
c'est ça "peuchère"
>>> s = ''' a ' \ " \n z '''
>>> s
' a \' \\ " \n z '
```

■TD4.4

On accède aux caractères individuels d'une chaîne de caractères par leur rang dans la chaîne, un caractère individuel étant vu lui-même comme une chaîne à un seul caractère.

```
>>> s = 'des caractères'
>>> s[9]
't'
>>> for c in s : print(c),
...
d e s   c a r a c t è r e s
```

```
>>> s[4:9]
'carac'
>>> s[len(s)-1]
's'
>>> s[:4] + 'mo' + s[9] + s[-1]
'des mots'
```

■TD4.5

Remarque 4.5 : *L'ordinateur stocke toutes les données sous forme numérique (ensemble de bits). En particulier, les caractères ont un équivalent numérique : le code ASCII (*American Standard Code for Information Interchange*). Le code ASCII de base code les caractères sur 7 bits (128 caractères : de 0 à 127, voir section 4.6.2 page 194).*
En PYTHON*, les fonctions standard* `ord` *et* `chr` *permettent de passer du caractère au code ASCII et inversement.*

```
>>> ord('e')          >>> chr(101)
101                   'e'
>>> ord('é')          >>> chr(233)
233                   'é'
```

TD 4.4 : OPÉRATIONS SUR LES CHAÎNES
Donner un exemple d'utilisation de chacune des opérations sur les chaînes de caractères décrites dans le tableau de la page 196.

TD 4.5 : INVERSER UNE CHAÎNE
Définir une fonction qui crée une copie d'une chaîne en inversant l'ordre des caractères.

```
>>> inverser('inverser')
'resrevni'
```

Les chaînes de caractères, et leurs opérations associées, sont évidemment très utiles pour manipuler du texte comme peut le faire un traitement de textes par exemple. ■TD4.6

TD 4.6 : Caractères, mots, lignes d'une chaîne
Définir une fonction qui compte le nombre de caractères, le nombre de mots et le nombres de lignes d'une chaîne de caractères.

4.2.3 Listes

Définition 4.8 : liste
Une liste est une séquence modifiable d'éléments.

Fig. 4.9 : Définitions de l'Académie (10)
LISTE *n. f. XVIe siècle. Emprunté de l'italien lista, de même sens. Suite ordonnée de noms, de mots, de chiffres, de nombres, de symboles, etc.*

D'un point de vue syntaxique, une liste est une suite d'éléments séparés par des virgules et encadrée par des crochets (`[...]`). On accède aux éléments individuels d'une liste comme pour une chaîne de caractères ou un n-uplet mais, contrairement aux chaînes et aux n-uplets, on peut modifier chaque élément d'une liste individuellement.

```
>>> s = [1,3,5,7]
>>> s[2]
5
>>> s[len(s)-1]
7
>>> for c in s : print(c),
...
1 3 5 7
```

```
>>> s[1:3]
[3,5]
>>> s[1:3] = [2,4]
>>> s
[1, 2, 4, 7]
>>> s[len(s):len(s)] = [8,9]
>>> s
[1, 2, 4, 7, 8, 9]
```

■TD4.7

TD 4.7 : Opérations sur les listes (1)
Donner un exemple d'utilisation de chacune des opérations sur les listes décrites dans le tableau de la page 195.

Pour créer une liste, on peut explicitement le faire *à la main*, comme ci-dessus avec l'instruction `s = [1,3,5,7]`, mais ce n'est raisonnable que pour de petites listes. On peut également utiliser la fonction prédéfinie `range` pour génèrer une liste d'entiers successifs (voir section 2.4.2 page 56). Si `range` est appelée avec un seul argument, la liste contiendra un nombre de valeurs égal à l'argument fourni, en commençant à partir de `0` (`range(n)` génère les nombres de `0` à `n-1` : `[0,1,2,...,n-1]`). On peut aussi utiliser `range` avec 2 ou 3 arguments : `range(debut,fin,pas)`, `debut` et `pas` étant optionnels. L'argument `debut` est la première valeur à générer (`debut = 0` par défaut), l'argument `pas` est l'incrément (`pas = +1` par défaut) qui permet de passer d'une valeur à la suivante (`[debut, debut + pas, debut + 2*pas,...]`) et l'argument obligatoire `fin` est la borne maximale (valeur `fin` exclue) des nombres à générer.

```
>>> range(5)
[0, 1, 2, 3, 4]
>>> range(2,5)
[2, 3, 4]
```

```
>>> range(0,5,2)
[0, 2, 4]
>>> range(5,0,-2)
[5, 3, 1]
```

La fonction `range` est fréquemment utilisée dans les boucles pour répéter un nombre de fois

connu à l'avance les instructions de la boucle. Mais d'une manière plus générale, on utilisera les fonctions d'ajout (`append`) ou d'insertion (`insert`) d'éléments pour créer de nouvelles listes par programme.

On dispose en effet pour les listes des opérations préconisées pour les collections (section 4.1.2 page 160) et que l'on retrouve dans le tableau de la page 195 :

- déterminer le nombre d'éléments de la liste `s`, `len(s)`
- tester l'appartenance d'un élément `x` à la liste `s`, `x in s`
- ajouter un élément `x` à la liste `s`, `s.append(x)`
  et plus généralement insérer un élément `x` au rang `i` dans la liste `s`, `s.insert(i,x)`
- supprimer l'élément de rang `i` de la liste `s`. `del(s[i])`

■TD4.8

On dispose également de quelques autres opérations particulièrement efficaces telles que compter les occurences (`s.count(x)`), retrouver le rang de la première occurence d'un élément (`s.index(x)`), inverser une liste (`s.reverse()`) ou trier une liste (`s.sort()`). ■TD4.9

```
>>> s = [1,2,1,1,2]
>>> s.count(1)
3
>>> s.index(2)
1
```

```
>>> s.reverse()
>>> s
[2, 1, 1, 2, 1]
>>> s.sort()
[1, 1, 1, 2, 2]
```

Comme nous l'avons déjà évoqué à l'occasion du passage des paramètres par référence lors de l'appel d'une fonction (section 3.3.1 page 115), le nom du liste est en fait une référence sur cette liste. Ainsi, si `s` est une liste, l'instruction `t = s` ne fait que copier la référence de la liste et non la liste elle-même : `t` devient un alias de `s` et toute modification d'un élément de `t` (ou de `s`) modifie également `s` (ou `t`). Pour copier la liste, il faudra d'une manière ou d'une autre copier les éléments un par un, comme ci-dessous à droite par exemple.

```
>>> s = [1,2,3]
>>> t = s
>>> t[0] = 9
>>> t
[9, 2, 3]
>>> s
[9, 2, 3]
```

```
>>> s = [1,2,3]
>>> t = []
>>> t[:0] = s[0:]
>>> t
[1, 2, 3]
>>> t[0] = 9
>>> t
[9, 2, 3]
>>> s
[1, 2, 3]
```

TD 4.8 : Opérations sur les listes (2)
Vérifier que les opérations suivantes sont équivalentes deux à deux :

```
del s[i:j]      et s[i:j] = []
s.append(x)     et s[len(s):len(s)] = [x]
s.extend(x)     et s[len(s):len(s)]= x
s.insert(i,x)   et s[i:i] = [x]
s.remove(x)     et del s[s.index(x)]
s.pop(i)        et x = s[i] ; del s[i] ; return x
```

TD 4.9 : Sélection d'éléments

1. *Définir une fonction qui crée une liste* `t` *composée des éléments d'une autre liste* `s` *qui vérifient une certaine condition* `p` *(*`p(s[i]) == True`*).*
2. *Définir une fonction qui supprime d'une liste* `s` *les éléments qui ne vérifient pas une certaine condition* `p`.

Remarque 4.6 : *En* Python*, l'instruction spéciale, appelée* list comprehension *:*

```
t = [expression for element in sequence]
```

est équivalente à :

```
t = []
for element in sequence :
    t.append(expression)
```

Exemples :

```
>>> s = [1,2,3]
>>> t = [c for c in s]
>>> t
[1, 2, 3]
```

```
>>> s = [1,2,3]
>>> y = [c%2 for c in s]
>>> y
[1, 0, 1]
```

4.2.4 Piles et files

Dans de nombreux cas, les seules opérations à effectuer sur les listes sont des insertions et des suppressions aux extrémités de la liste. C'est le cas en particulier des piles (*stack*) et des files (*queue*).

Exemple 4.5 : Pile d'assiettes
Dans une pile d'assiettes (figure 4.11), on ajoute des assiettes au sommet de la pile, et on les récupère dans l'ordre inverse, en commençant par la dernière ajoutée.

Définition 4.9 : pile
Une pile est une séquence dans laquelle on ne peut ajouter et supprimer un élément qu'à une seule extrémité : le « sommet » de la pile.

Dans le cas des piles, on ajoute (on empile) et on supprime (on dépile) à une seule extrémité. Une pile est ainsi basée sur le principe « dernier arrivé, premier sorti » (Lifo : *Last In, First Out*) : le dernier élément ajouté au sommet de la pile sera le premier à être récupéré. Les seules opérations nécessaires pour une pile sont ainsi :

- tester si la pile `p` est vide, `ok = emptyStack(p)`
- accéder au sommet `x` de la pile `p`, `x = topStack(p)`
- empiler un élément `x` au sommet de la pile `p`, `pushStack(p,x)`
- dépiler l'élément `x` qui se trouve au sommet de la pile `p`. `x = popStack(p)`

■ TD4.10

Exemple 4.6 : File d'attente de voitures
A un péage (figure 4.12), la première voiture à être entrée dans une file d'attente sera la première sortie de la file.

Définition 4.10 : file
Une file est une séquence dans laquelle on ne peut ajouter un élément qu'à une seule extrémité et ne supprimer un élément qu'à l'autre extrémité : la « tête » de la file.

Dans le cas des files, on ajoute (on enfile) à une extrémité et on supprime (on défile) à l'autre extrémité. Une file est ainsi basée sur le principe « premier arrivé, premier sorti » (Fifo : *First In, First Out*) : le premier élément ajouté sera le premier à être récupéré en tête de file.

Fig. 4.10 : Définitions de l'Académie (11)
FILE *n. f. XVe siècle. Déverbal de filer. Suite de personnes ou de choses placées les unes derrière les autres sur une même ligne.*

PILE *n. f. Ensemble d'objets de même sorte placés les uns sur les autres.*

Fig. 4.11 : Piles d'assiettes

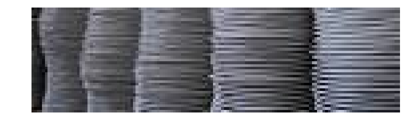
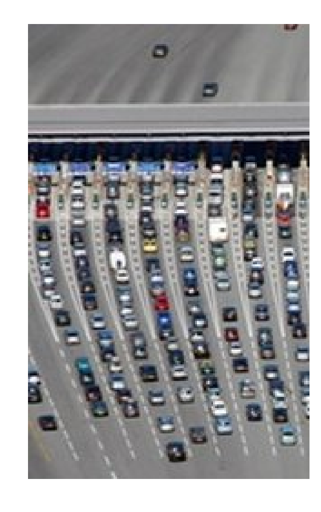


Fig. 4.12 : Files d'attente à un péage



Les seules opérations nécessaires pour une file sont ainsi :

- tester si la file `f` est vide, `ok = emptyQueue(f)`
- accéder à la tête `x` de la file `f`, `x = topQueue(f)`
- enfiler un élément `x` dans la pile `f`, `pushQueue(f,x)`
- dépiler l'élément `x` qui se trouve en tête de la file `f`. `x = popQueue(f)`

■TD4.11

TD 4.10 : Opérations sur les piles
Définir les 4 opérations sur les piles définies ci-contre : `emptyStack`, `topStack`, `pushStack` *et* `popStack`. *On empilera et dépilera à la fin de la liste qui sert à stocker les éléments de la pile.*

TD 4.11 : Opérations sur les files
Définir les 4 opérations sur les files définies ci-contre : `emptyQueue`, `topQueue`, `pushQueue` *et* `popQueue`. *On enfilera en début de liste et on défilera à la fin de la liste qui sert à stocker les éléments de la file.*

4.2.5 Listes multidimensionnelles

Un autre cas particulier de liste est celui où les éléments d'une liste sont eux-mêmes des listes. On parle alors de listes de listes (ou de tableaux de tableaux) ou de listes multidimensionnelles (ou de tableaux multidimensionnels). Un damier, une image numérique ou encore une matrice (figure 4.13) sont des exemples de tableaux bidimensionnels.

Fig. 4.13 : Définitions de l'Académie (12)
MATRICE *n. f. XIIIe siècle. Emprunté du latin matrix, matricis, « reproductrice », dérivé de mater, « mère ». MATH. Ensemble ordonné de nombres algébriques, présenté sous forme de tableau. Les dimensions d'une matrice, le nombre de lignes et de colonnes qui constituent le tableau. Matrice carrée, qui a le même nombre de lignes et de colonnes.*

Dans l'exemple ci-dessous, la liste `s` est composé de 3 listes (`len(s) == 3`) de longueurs différentes. Un premier niveau de crochets donne accès à un élément de la liste `s` (comme `s[2]` par exemple) : cet élément est lui-même une liste (la liste `[6,7,8,9]`) ; un deuxième niveau de crochets donne accès à un élément de cette liste (comme `s[2][1]` par exemple). D'une manière générale, il faudra autant de niveaux de crochets que de listes imbriquées pour accéder aux éléments individuels.

```
>>> s = [[4,5],[1,2,3],[6,7,8,9]]
>>> type(s)
<type 'list'>
>>> len(s)
3
>>> type(s[2])
<type 'list'>
```

```
>>> s[2]
[6, 7, 8, 9]
>>> s[2][1]
7
>>> s[1][2]
3
```

Fig. 4.14 : Echiquier



Exemple 4.7 : Echiquier
L'échiquier de la figure 4.14 ci-contre est un ensemble de 8 lig nes (numérotées de `1` *à* `8` *sur la figure) composées chacune de 8 cases (nommées de* `A` *à* `H`*). Il peut être représenté par une liste* `s` *de 8 listes, chacune composée de 8 éléments. Ainsi, le roi blanc (en* `8G` *sur la figure) sera l'élément* `s[7][6]` *et le cavalier blanc (en* `4D`*) sera l'élément* `s[3][3]` *de la liste* `s` *en tenant compte de la convention informatique qui fait débuter à* `0` *les indices d'une liste.*

Le parcours d'un tableau bidimensionnel nécessite 2 boucles imbriquées : la première pour

accéder aux lignes les unes après les autres, la deuxième pour accéder aux éléments individuels de chaque ligne.

```
>>> s = [[4,5],[1,2,3],[6,7,8,9]]
>>> for c in s : print(c)
...
[4, 5]
[1, 2, 3]
[6, 7, 8, 9]
```

```
>>> s[2]
[6, 7, 8, 9]
>>> for c in s :
...     for e in c : print(e,end=' ')
...     print()
...
4 5
1 2 3
6 7 8 9
```

D'une manière générale, il faudra autant de boucles imbriquées qu'il y a de listes imbriquées pour parcourir l'ensemble des éléments individuels.

```
>>> s = [[[4,5],[1,2,3]], [[0],[10,11],[6,7,8,9]]]
>>> len(s)
2
>>> len(s[1])
3
>>> len(s[1][2])
4
>>> s[1][2][2]
8
>>> for c in s : print(c)
...
[[4, 5], [1, 2, 3]]
[[0], [10, 11], [6, 7, 8, 9]]
```

```
>>> for c in s :
...     for e in c : print(e),
...     print()
...
[4, 5] [1, 2, 3]
[0] [10, 11] [6, 7, 8, 9]
>>> for c in s :
...     for e in c :
...         for x in e :
print(x,end=' ')
...     print()
...
4 5 1 2 3
0 10 11 6 7 8 9
```

Un cas particulier important de liste multidimensionnelle est la notion de matrice. En mathématiques, les matrices servent à interpréter en termes calculatoires et donc opérationnels les résultats théoriques de l'algèbre linéaire [8]. On représente généralement une matrice A sous la forme d'un tableau rectangulaire de n lignes et de m colonnes :

Remarque 4.7 : *Les n lignes d'une matrice de dimensions (n,m) ont toutes la même longueur m contrairement au cas plus général des listes multidimensionnelles qui peuvent avoir des lignes de différentes longueurs.*

$$A = (a_{ij}) = \begin{pmatrix} a_{00} & a_{01} & \cdots & a_{0(m-2)} & a_{0(m-1)} \\ a_{10} & a_{11} & \cdots & a_{1(m-2)} & a_{1(m-1)} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ a_{(n-1)0} & a_{(n-1)1} & \cdots & a_{(n-1)(m-2)} & a_{(n-1)(m-1)} \end{pmatrix}$$

La matrice est dite carrée quand $n = m$ et symétrique quand $\forall i, j\ a_{ij} = a_{ji}$.

En informatique, une matrice sera simplement représentée par un tableau bidimensionnel comme la matrice de dimensions $(n = 2, m = 3)$ ci-dessous.

```
>>> s = [[1,2,3],[4,5,6]]
>>> n = len(s)
>>> m = len(s[0])
>>> len(s[0]) == len(s[1])
True
```

```
>>> n
2
>>> m
3
```

TD 4.12 : Produit de matrices
Définir la fonction qui calcule la matrice C, produit de 2 matrices A et B respectivement de dimensions (n, r) et (r, m).

$$c_{ij} = \sum_{k=0}^{r-1} a_{ik} \cdot b_{kj}$$

B

A

C

Exemple 4.8 : Addition de 2 matrices
L'addition de 2 matrices A et B de mêmes dimensions (n, m) donne une nouvelle matrice C de dimensions (n, m) telle que $c_{ij} = a_{ij} + b_{ij}$.

```
def additionMatrice(a,b):
    n, m = len(a), len(a[0])
    s = []
    for i in range(n):
        c = []
        for j in range(m):
            c.append(a[i][j]+b[i][j])
        s.append(c)
    return s
```

```
>>> a = [[1,2,3],[4,5,6]]
>>> b = [[-1,-2,-3],[-1,-2,-3]]
>>> additionMatrice(a,b)
[[0, 0, 0], [3, 3, 3]]
>>> additionMatrice(a,a)
[[2, 4, 6], [8, 10, 12]]
```

■ TD4.12

Les matrices sont utilisées pour de multiples applications et servent notamment à représenter les coefficients des systèmes d'équations linéaires (voir TD 4.29 page 184 et annexe 4.6.5 page 199).

4.3 Recherche dans une séquence

Remarque 4.8 : *L'opération de recherche est tellement importante qu'elle est aujourd'hui toujours disponible dans les systèmes d'exploitation et dans les langages de programmation. En* Python*, la méthode* `index` *effectue cette recherche pour les chaînes et pour les listes. Il est donc rarement nécessaire de la rédéfinir soi-même ; nous le faisons ici pour des raisons pédagogiques.*

En informatique, une des opérations essentielles est de retrouver une information stockée en mémoire vive, sur un disque dur ou quelque part sur le réseau. Il suffit pour s'en convaincre de penser au nombre de fois dans la journée où l'on recherche un numéro de téléphone sur son portable ou une information quelconque sur internet à l'aide d'un moteur de recherche.

Nous nous intéresserons ici à 2 méthodes de recherche d'un élément dans une liste : la recherche séquentielle et la recherche dichotomique.

4.3.1 Recherche séquentielle

La recherche séquentielle d'un élément `x` dans une liste `t` consiste à comparer l'élément recherché `x` successivement à tous les éléments de la liste `t` jusqu'à trouver une correspondance. Autrement dit, on compare l'élément recherché `x` au premier élément de la liste `t`. S'ils sont identiques, on a trouvé le rang de la première occurence de `x` dans `t` ; sinon, on recommence la recherche à partir du deuxième élément. Et ainsi de suite... On en déduit la version récursive ci-dessous généralisée à une recherche entre deux rangs `debut` et `fin` de la liste `t`. Dans cette version, nous avons choisi de retourner un couple de valeurs `(ok,r)` : `ok` est un booléen qui teste si la recherche est fructueuse ou non et `r` est le rang de la première occurence de l'élément recherché s'il a effectivement été trouvé.

```
def recherche(t,x,debut,fin):
    ok,r = False,debut
    if r > fin: ok = False
    else:
        if t[r] == x: ok = True
        else: ok,r = recherche(t,x,r+1,fin)
    return ok,r
```

```
>>> s = [1,3,5,6,5,2]
>>> recherche(s,5,0,len(s)-1)
(True, 2)
>>> recherche(s,5,3,len(s)-1)
(True, 4)
>>> recherche(s,4,0,len(s)-1)
(False, 6)
```

■TD4.13

La version itérative équivalente est donnée ci-dessous en précisant les préconditions d'utilisation de l'algorithme.

Recherche séquentielle

```
def rechercheSequentielle(t,x,debut,fin):
    assert type(t) is list
    assert 0 <= debut <= fin < len(t)
    ok, r = False, debut
    while r <= fin and not ok:
        if t[r] == x: ok = True
        else: r = r + 1
    return ok,r
```

Le nombre d'itérations effectuées pour retrouver `x` est égal à la longueur n de la liste `t` si `x` n'est pas dans la liste et à `r`, rang de la première occurence de `x`, si `x` est dans la liste. Ainsi, si on définit la complexité c de cet algorithme comme le nombre d'itérations effectuées, elle est

TD 4.13 : Annuaire téléphonique

On considère un annuaire téléphonique stocké sous la forme d'une liste de couples `(nom,téléphone)` *(exemple :* `[('jean','0607080910'),('paul','0298000102')]`*).*

1. *Définir une fonction qui retrouve dans un annuaire téléphonique le numéro de téléphone à partir du nom.*
2. *Définir la fonction inverse qui retrouve le nom à partir du numéro de téléphone.*

Remarque 4.9 : *On définit la complexité d'un algorithme A comme une fonction $f_A(n)$ de la taille n des données. Pour analyser cette complexité, on s'attache à déterminer l'ordre de grandeur asymptotique de $f_A(n)$: on cherche une fonction connue qui a une rapidité de croissance voisine de celle de $f_A(n)$. On utilise pour cela la notation mathématique $f = O(g)$, qui se lit « f est en grand O de g », et qui indique avec quelle rapidité une fonction « augmente » ou « diminue ». Ainsi $f = O(g)$ signifie que l'ordre de grandeur asymptotique de f est inférieur ou égal à celui de g.*

Notation	*Type de complexité*
$O(1)$	*complexité constante*
$O(\log(n))$	*complexité logarithmique*
$O(n)$	*complexité linéaire*
$O(n\log(n))$	*complexité quasi-linéaire*
$O(n^2)$	*complexité quadratique*
$O(n^3)$	*complexité cubique*
$O(n^p)$	*complexité polynomiale*
$O(n^{\log(n)})$	*complexité quasi-polynomiale*
$O(2^n)$	*complexité exponentielle*
$O(n!)$	*complexité factorielle*

Remarque 4.10 : *Sauf mention contraire, une liste triée le sera ici selon l'ordre croissant de ses éléments.*

TD 4.14 : Recherche dichotomique
La recherche dichotomique présentée ci-contre n'assure pas de trouver la première occurence d'un élément `x` *dans une liste* `t` *triée.*
Modifier l'algorithme de recherche dichotomique proposé pour rechercher la première occurence d'un élément `x` *dans une liste* `t` *triée.*

au pire égale à n, au mieux égale à 1. En moyenne, on peut démontrer (voir [9] ou [12]) qu'elle vaut $(1-q)\cdot n + (n+1)\cdot q/2$ si q est la probabilité que `x` soit dans `t`, quelle que soit sa position dans la liste (toutes les places sont équiprobables). On retrouve bien $c = n$ si $q = 0$. On a également $c = (n+1)/2$ si $q = 1$ et $c = (3n+1)/4$ si $q = 1/2$. Dans tous les cas, l'ordre de grandeur de la complexité de la recherche séquentielle est n, le nombre d'éléments dans la liste : on parle alors de complexité linéaire, notée $O(n)$. Il s'agit d'une recherche linéaire en fonction du nombre d'éléments dans la liste : si le nombre d'éléments est multiplié par 100, le complexité sera multipliée par 100 en moyenne, et donc le temps d'exécution de la recherche également.

4.3.2 Recherche dichotomique

Dans certains cas, la recherche d'un élément `x` s'effectue dans une liste `t` triée. La méthode qui suit, dite recherche par dichotomie ou recherche dichotomique, utilise pleinement le fait que les éléments de la liste sont triés.

Le principe de la recherche dichotomique consiste à comparer `x` avec l'élément `t[m]` du milieu de la liste `t` :

- si `x == t[m]`, on a trouvé une solution et la recherche s'arrête ;
- si `x < t[m]`, `x` ne peut se trouver que dans la moitié gauche de la liste `t` puisque celle-ci est triée par ordre croissant ; on poursuit alors la recherche de la même manière uniquement dans la moitié gauche de la liste ;
- si `x > t[m]`, `x` ne peut se trouver que dans la moitié droite de la liste `t` ; on poursuit alors la recherche uniquement dans la moitié droite de la liste.

A chaque fois, on poursuit donc la recherche en diminuant de moitié le nombre d'éléments restant à traiter. En suivant ce principe de recherche, on est naturellement amené à écrire une version récursive de l'algorithme de recherche dichotomique.

```
def dichotomie(t,x,gauche,droite):
    ok, m = False, (gauche + droite)/2
    if gauche > droite: ok = False
    else:
        if t[m] == x: ok = True
        elif t[m] > x:
            ok,m = dichotomie(t,x,gauche,m-1)
        else:
            ok,m = dichotomie(t,x,m+1,droite)
    return ok,m
```

```
>>> s = [1,3,5,6,6,9]
>>> dichotomie(s,6,0,len(s)-1)
(True, 4)
>>> dichotomie(s,6,2,len(s)-1)
(True, 3)
>>> dichotomie(s,4,0,len(s)-1)
(False, 1)
```

■ TD4.14

La version itérative équivalente est donnée ci-dessous en précisant les préconditions d'utilisation de l'algorithme.

Remarque 4.11 : *On peut montrer (voir [9] ou [12]) que la complexité de l'algorithme de recherche dichotomique est de l'ordre de $\log(n)$ où n est la longueur de la liste. Il s'agit d'une complexité logarithmique (voir remarque 4.9 page 172).*

Recherche dichotomique

```
def rechercheDichotomique(t,x,gauche,droite):
    assert type(t) is list
    assert 0 <= gauche <= droite < len(t)
    ok, m = False, (gauche + droite)/2
    while gauche <= droite and not ok:
        m = (gauche + droite)/2
        if t[m] == x: ok = True
        elif t[m] > x: droite = m - 1
        else: gauche = m + 1
    return ok,m
```

4.4 Tri d'une séquence

Pour retrouver une information, il est souvent plus efficace que les données soient préalablement triées : c'est ce que nous avons constaté avec la recherche par dichotomie de la section précédente. Là encore, pour s'en convaincre, il suffit de penser à la recherche d'un mot dans un dictionnaire ou d'un nom dans un bottin.

Dans la suite, nous supposerons qu'il existe une relation d'ordre total, entre les éléments de la liste, notée $\leq$ et qui vérifie les propriétés habituelles de réflexivité, d'antisymétrie et de transitivité :

1. réflexivité : $x \leq x$
2. antisymétrie : $(x \leq y)$ `and` $(y \leq x) \Rightarrow x = y$
3. transitivité : $(x \leq y)$ `and` $(y \leq z) \Rightarrow (x \leq z)$

La fonction récursive `enOrdre` ci-dessous teste si la liste `t` est triée par ordre croissant de ses éléments entre les rangs `debut` et `fin`, en utilisant la relation d'ordre `<=` (`t[debut] <= t[debut+1]`).

```
def enOrdre(t,debut,fin):
    assert type(t) is list
    assert 0 <= debut <= fin < len(t)
    ok = False
    if debut == fin: ok = True
    else:
        if t[debut] <= t[debut+1]:
            ok = enOrdre(t,debut+1,fin)
    return ok
```

```
>>> s = [3,1,2,3,1]
>>> enOrdre(s,0,len(s)-1)
False
>>> enOrdre(s,1,3)
True
>>> enOrdre(s,1,4)
False
```

■ TD4.15

TD 4.15 : Liste ordonnée
Définir une version itérative de la fonction récursive `enOrdre` *définie ci-contre. On appliquera la méthode de transformation décrite en section 3.3.4 page 127.*

Nous nous intéresserons ici à 3 méthodes de tri d'une liste : le tri par sélection, le tri par insertion et le tri rapide.

4.4.1 Tri par sélection

Le tri par sélection d'une liste consiste à rechercher le minimum de la liste à trier, de le mettre en début de liste en l'échangeant avec le premier élément et de recommencer sur le reste de la liste (figure 4.16). La version récursive ci-dessous trie la liste `t` par ordre croissant entre les rangs `debut` et `fin` ; elle fait appel à la fonction `minimum` qui retourne le rang du plus petit élément de la séquence `t` entre les rangs `debut` et `fin`.



Fig. 4.15 : Tri par sélection

6	4	1	3	5	2
1	4	**6**	3	5	2
1	**2**	6	3	5	**4**
1	2	**3**	**6**	5	4
1	2	3	**4**	5	**6**
1	2	3	4	**5**	6
1	2	3	4	5	**6**

Tri de la liste `[6,4,1,3,5,2]` *: en gras, les valeurs échangées à chaque étape.*

```
def triSelection(t,debut,fin):
    assert type(t) is list
    assert 0 <= debut <= fin < len(t)
    if debut < fin:
        mini = minimum(t,debut,fin)
        t[debut],t[mini] = t[mini],t[debut]
        triSelection(t,debut+1,fin)
    return
```

```
def minimum(t,debut,fin):
    assert type(t) is list
    assert 0 <= debut <= fin < len(t)
    mini = debut
    for j in range(debut+1,fin+1):
        if t[j] < t[mini]: mini = j
    return mini
```

```
>>> s = [5,4,3,2,1,0]
>>> triSelection(s,0,len(s)-1)
>>> s
[0, 1, 2, 3, 4, 5]
```

```
>>> s = [5,4,3,2,1,0]
>>> minimum(s,0,len(s)-1)
5
>>> s[minimum(s,0,len(s)-1)]
0
>>> s[minimum(s,0,2)]
3
```

■ TD4.16

TD 4.16 : Tri d'un annuaire téléphonique
On considère un annuaire téléphonique stocké sous la forme d'une liste de couples `(nom,téléphone)` *(exemple :* `[('paul','0607080910'),('jean','0298000102')]`*). Définir une fonction qui trie un annuaire téléphonique par ordre alphabétique des noms.*

On en déduit la version itérative équivalente :

Tri par sélection

```
def triSelection(t,debut,fin):
    assert type(t) is list
    assert 0 <= debut <= fin < len(t)
    while debut < fin:
        mini = minimum(t,debut,fin)
        t[debut],t[mini] = t[mini],t[debut]
        debut = debut + 1
    return
```

■ TD4.17

TD 4.17 : Complexité du tri par sélection
Montrer que la complexité du tri par sélection est :

1. *en $O(n)$ en ce qui concerne les échanges de valeurs au sein d'une liste de longueur n,*
2. *en $O(n^2)$ en ce qui concerne les comparaisons entre éléments de la liste.*

4.4.2 Tri par insertion

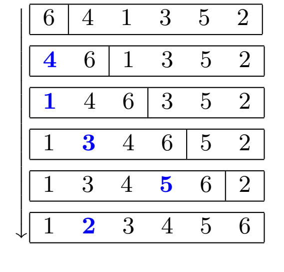
Dans le tri par insertion, on trie successivement les premiers éléments de la liste : à la $i^{ème}$ étape, on insère le $i^{ème}$ élément à son rang parmi les $i-1$ éléments précédents qui sont déjà triés entre eux. Dans la version récursive de la fonction `triInsertion`, on insère l'élément `t[fin]` dans la liste des éléments précédents déjà triée de `debut` à `(fin - 1)`.

```
def triInsertion(t,debut,fin):
    assert type(t) is list
    assert 0 <= debut <= fin < len(t)
    if fin > debut:
        triInsertion(t,debut,fin-1)
        i = fin - 1
        x = t[fin]
        while i >= debut and t[i] > x:
            t[i+1] = t[i]
            i = i - 1
        t[i+1] = x
    return
```

```
>>> s = [9,8,7,6,5,4]
>>> triInsertion(s,0,len(s)-1)
>>> s
[4, 5, 6, 7, 8, 9]
>>> s = [9,8,7,6,5,4]
>>> triInsertion(s,1,4)
>>> s
[9, 5, 6, 7, 8, 4]
```

La procédure précédente n'est pas récursive terminale et on ne peut donc pas utiliser la méthode de la section 3.3.4 page 127 pour la transformer en procédure itérative. Considérons alors une procédure récursive non terminale du type :


Fig. 4.16 : Tri par insertion



Tri de la liste `[6,4,1,3,5,2]` *: en gras, les valeurs insérées dans la partie gauche à chaque étape.*

Remarque 4.12 : *Le tri par insertion correspond à la stratégie d'un joueur de cartes qui trie son jeu.*

```
def f(x):
    instruction_1
    f(g(x))
    instruction_2
    return
```

`x` représente ici la liste des arguments de la fonction, `instruction_1` et `instruction_2` des blocs d'instructions qui encadrent l'appel récursif `f(g(x))` et `g(x)` une transformation des arguments de `f`. La fonction `triInsertion` est bien de ce type : `x = t,debut,fin`, `instruction_1` est vide, `g(x)` : `t,debut,fin` → `t,debut,fin-1` et `instruction_2` regroupe toutes les instructions qui suivent l'appel récursif.

La méthode consiste à sauvegarder, à la fin de l'exécution de `instruction_1`, la valeur de `x` sur une pile (voir section 4.2.4 page 167). Puis on exécute l'appel récursif, et on restaure la valeur de `x` (on dépile `x`) avant d'exécuter `instruction_2`. Pour un appel `triInsertion(t,debut,fin)`, on va empiler successivement `fin`, `fin-1`, `fin-2`, . . . ,`debut+1` (seul `fin` évolue lors des appels), puis on récupèrera sur la pile les valeurs en sens inverse `debut+1`, `debut+2`, . . . ,`fin-2`, `fin-1` et `fin` pour chacune desquelles on exécutera la fin de la procédure. Une simple boucle `for k in range(debut+1,fin+1)` permettra de simuler cette récupération.

La version itérative ci-dessous est équivalente à la procédure récursive. On retrouve des lignes 5 à 10 les instructions qui suivent l'appel récursif de la version récursive ; la boucle de la ligne 4 simule la récupération des valeurs successives de l'argument `fin` sauvegardées avant l'appel récursif.

TD 4.18 : Tri par insertion
Exécuter à la main l'appel `triInsertion([9,8,7,6,5,4],1,4)` *en donnant les valeurs des variables* `i`, `k`, `x` *et* `t` *à la fin de chaque itération de la boucle* `while` *(ligne 7 de la version itérative de l'algorithme).*

Tri par insertion

```
1  def triInsertion(t,debut,fin):
2      assert type(t) is list
3      assert 0 <= debut <= fin < len(t)
4      for k in range(debut+1,fin+1):
5          i = k - 1
6          x = t[k]
7          while i >= debut and t[i] > x:
8              t[i+1] = t[i]
9              i = i - 1
10         t[i+1] = x
11     return
```

■ TD4.18

On peut démontrer que la complexité du tri par insertion est en $O(n^2)$ aussi bien en ce qui concerne les échanges de valeurs au sein de la liste que les comparaisons entre éléments de la liste.

4.4.3 Tri rapide

Les complexités des tris par sélection et par insertion sont, soit en nombre d'échanges de valeurs, soit en nombre de comparaisons entre éléments, quadratiques en $O(n^2)$. La méthode de tri rapide (*quicksort*) présentée ici a une complexité quasi-linéaire en $O(n \log(n))$.

Le principe du tri rapide est le suivant : on partage la liste à trier en deux sous-listes telles que tous les éléments de la première soient inférieurs à tous les éléments de la seconde. Pour partager la liste en deux sous-listes, on choisit un des éléments de la liste (par exemple le premier) comme pivot. On construit alors une sous-liste avec tous les éléments inférieurs ou égaux à ce pivot et une sous-liste avec tous les éléments supérieurs au pivot (figure 4.17). On trie les deux sous-listes selon le même processus jusqu'à avoir des sous-listes réduites à un seul élément (figure 4.18).

```
def triRapide(t,debut,fin):
    assert type(t) is list
    assert 0 <= debut
    assert fin <= len(t)
    if debut < fin:
        pivot = t[debut]
        place = partition(t,debut,fin,pivot)
        triRapide(t,debut,place-1)
        triRapide(t,place+1,fin)
    return

>>> s = [9,8,7,6,5,4]
>>> triRapide(s,0,len(s)-1)
>>> s
[4, 5, 6, 7, 8, 9]
>>> s = [9,8,7,6,5,4]
>>> triRapide(s,1,4)
>>> s
[9, 5, 6, 7, 8, 4]
```

```
def partition(t,debut,fin,pivot):
    p,inf,sup = debut,debut,fin;
    while p <= sup:
        if t[p] == pivot:
            p = p + 1
        elif t[p] < pivot:
            t[inf],t[p] = t[p],t[inf]
            inf = inf + 1
            p = p + 1
        else:
            t[sup],t[p] = t[p],t[sup]
            sup = sup - 1
    if p > 0: p = p - 1
    return p

>>> s = [3,5,2,6,1,4]
>>> partition(s,0,len(s)-1,s[0]), s
(2, [1, 2, 3, 6, 5, 4])
```

Pour partitionner la liste, on a utilisé 2 compteurs `inf` et `sup` qui partent des 2 extrémités de la liste en évoluant l'un vers l'autre. Le compteur de gauche `inf` part du début de la liste et lorsqu'il atteint un élément supérieur au pivot, on arrête sa progression. De même, le compteur de droite `sup` part de la fin de la liste et s'arrête lorsqu'il atteint un élément inférieur au pivot. On échange alors ces deux éléments (`t[sup],t[inf] = t[inf],t[sup]`) puis on continue à faire progresser les compteurs et à faire des échanges jusqu'à ce que les compteurs se croisent.

Ainsi, le tri rapide correspond au schéma récursif suivant :

Remarque 4.13 : *Le tri rapide est une méthode de tri inventée par Charles Antony Richard Hoare en 1961 et basée sur le principe* diviser pour régner. *C'est en pratique un des tris sur place le plus rapide pour des données réparties aléatoirement.*

Fig. 4.17 : Partition d'une liste

$t_i \leq pivot$	$pivot$	$t_i > pivot$

La partition d'une liste est une variante du problème du drapeau hollandais dû à l'informaticien néerlandais Edsger Wybe Dijkstra. *Dans sa version originale, on remplit aléatoirement un tableau avec les couleurs bleu, blanc et rouge. Le problème consiste à trouver un algorithme qui, par des échanges successifs de composantes, reconstitue le drapeau hollandais. L'algorithme ne doit pas utiliser de tableau intermédiaire et ne parcourir le tableau qu'une seule fois.*

Fig. 4.18 : Tri rapide

6	4	1	3	5	2
2	4	1	3	5	**6**
1	**2**	4	3	5	6
1	2	3	**4**	5	6
1	2	3	4	5	6

Tri de la liste `[6,4,1,3,5,2]` *: en gras, les valeurs des pivots à chaque étape.*

```
def f(x):
    if cond:
        instructions
        f(g(x))
        f(h(x))
    return
```

`x` représente ici la liste des arguments de la fonction, `cond` une condition portant sur `x`, `instructions` un bloc d'instructions qui précède les 2 appels récursifs `f(g(x))` et `f(h(x))`. `g(x)` et `h(x)` sont des transformations des arguments de `f`. Dans le cas du tri rapide, on a `g(x)` : `fin = place - 1` et `h(x)` : `debut = place + 1`.

On peut supprimer le 2ème appel récursif (`f(h(x))`), qui est terminal, selon la méthode exposée en section 3.3.4 page 127. Ce qui donne :

```
def f(x):
    while cond:
        instructions
        f(g(x))
        x = h(x)
    return
```

```
def triRapide(t,debut,fin):
    assert type(t) is list
    assert 0 <= debut
    assert fin <= len(t)
    while debut < fin:
        pivot = t[debut]
        place = partition(t,debut,fin,pivot)
        triRapide(t,debut,place-1)
        debut = place + 1
    return
```

La suppression du 1er appel récursif (`f(g(x))`) est moins immédiate car cet appel n'est pas terminal et la valeur des arguments `x` ne varie pas de manière aussi prévisible que pour le tri par insertion (voir section 4.4.2 page 175). Comme suggéré en section 4.4.2 page 176, on peut utiliser une pile pour sauvegarder les contextes d'appel de la première récursivité (`empiler(pile,x)`) avant de les restaurer pour la deuxième récursivité (`x = depiler(pile)`) :

```
def f(x):
    pile = []
    while True:
        while cond:
            instructions
            empiler(pile,x)
            x = g(x)
        if not len(pile) == 0:
            x = depiler(pile)
            x = h(x)
        else: return
    return
```

```
def empiler(p,x):
    assert type(p) is list
    p.append(x)
    return

def depiler(p):
    assert type(p) is list
    assert len(p) > 0
    x = p[len(p)-1]
    del p[len(p)-1]
    return x
```

Ce qui donne la version itérative suivante pour le tri rapide :

```
def triRapide(t,debut,fin):
    assert type(t) is list
    assert 0 <= debut
    assert fin <= len(t)
    pile = []
    while True:
        while debut < fin:
            pivot = t[debut]
            place = partition(t,debut,fin,pivot)
            empiler(pile,(debut,fin,place))
            fin = place - 1
        if not len(pile) == 0:
            (debut,fin,place) = depiler(pile)
            debut = place + 1
        else: return
    return
```

```
>>> s = [9,8,7,6,5,4]
>>> triRapide(s,0,len(s)-1)
>>> s
[4, 5, 6, 7, 8, 9]
>>> s = [9,8,7,6,5,4]
>>> triRapide(s,1,4)
>>> s
[9, 5, 6, 7, 8, 4]
```

A notre niveau, on retiendra cependant la version récursive, plus proche de la description du tri rapide :

Tri rapide

```
def triRapide(t,debut,fin):
    assert type(t) is list
    assert 0 <= debut
    assert fin <= len(t)
    if debut < fin:
        pivot = t[debut]
        place = partition(t,debut,fin,pivot)
        triRapide(t,debut,place-1)
        triRapide(t,place+1,fin)
    return
```

TD4.19

TD 4.19 : Comparaison d'algorithmes (1)
Comparer les temps d'exécution des versions récursive et itérative du tri rapide pour différentes tailles de liste.
On pourra utiliser la fonction `random()` *ou* `randint(min,max)` *du module standard* `random` *pour générer des listes de nombres aléatoires, et la fonction* `time()` *du module standard* `time` *pour mesurer le temps avant et après l'appel des fonctions.*

4.5 Exercices complémentaires

4.5.1 Connaître

TD 4.20 : QCM (4)

(un seul item correct par question)

1. *Le type d'une variable définit*
   (a) *l'ensemble des noms qu'elle peut prendre.*
   (b) *le regroupement fini des données qui la composent et dont le nombre n'est pas fixé* a priori.
   (c) *l'ensemble des valeurs qu'elle peut prendre et l'ensemble des opérations qu'elle peut subir.*
   (d) *le regroupement hiérarchique des données qui la composent*
2. *Une séquence est*
   (a) *un regroupement fini de données dont le nombre n'est pas fixé* a priori.
   (b) *une collection d'éléments, appelés « sommets », et de relations entre ces sommets.*
   (c) *une collection non ordonnée d'éléments.*
   (d) *une suite ordonnée d'éléments accessibles par leur rang dans la séquence.*
3. *Parmi les exemples suivants, le seul exemple de séquence est*
   (a) *le tableau final d'un tournoi de tennis.*
   (b) *la carte routière.*
   (c) *la classification des espèces animales.*
   (d) *la main au poker.*
4. *Parmi les types suivants, un seul n'est pas une variante de séquence. Lequel ?*
   (a) *le n-uplet.*
   (b) *la chaîne de caractères.*
   (c) *le dictionnaire.*
   (d) *la pile.*
5. *Dans une liste*

   (a) *tous les éléments sont du même type.*
   (b) *les éléments peuvent avoir des types différents.*
   (c) *les éléments ne peuvent pas être des dictionnaires.*
   (d) *les éléments ont comme type un des types de base (*`bool`*,*`int`*,*`float`*).*

6. *Dans la liste multidimensionnelle* `s = [[1,2,3,4],[5,6,7],[8,9]]` *que vaut* `s[1][1]` *?*
   (a) *1.*
   (b) *2.*
   (c) *5.*
   (d) *6.*
7. *Une pile est une séquence dans laquelle*
   (a) *on ne peut ajouter un élément qu'à une seule extrémité et ne supprimer un élément qu'à l'autre extrémité.*
   (b) *on ne peut ajouter un élément qu'à une seule extrémité et en supprimer n'importe où.*
   (c) *on ne peut ajouter et supprimer un élément qu'à une seule extrémité.*
   (d) *on ne peut supprimer un élément qu'à une seule extrémité et en ajouter n'importe où.*
8. *Une file est une séquence dans laquelle*
   (a) *on ne peut ajouter un élément qu'à une seule extrémité et ne supprimer un élément qu'à l'autre extrémité.*
   (b) *on ne peut ajouter un élément qu'à une seule extrémité et en supprimer n'importe où.*
   (c) *on ne peut ajouter et supprimer un élément qu'à une seule extrémité.*
   (d) *on ne peut supprimer un élément qu'à une seule extrémité et en ajouter n'importe où.*
9. *La recherche séquentielle d'un élément dans une liste consiste à*
   (a) *rechercher le minimum de la liste et à le mettre en début de liste en l'échangeant avec cet élément.*
   (b) *rechercher le maximum de la liste et à le mettre en début de liste en l'échangeant avec cet élément.*
   (c) *comparer l'élément recherché successivement à tous les éléments de la liste jusqu'à trouver une correspondance.*

(d) *comparer l'élément recherché avec l'élément milieu de la liste et poursuivre de même dans la sous-liste de droite ou dans la sous-liste de gauche à l'élément milieu.*

10. *Dans le tri par insertion*

    (a) *on partage la liste à trier en deux sous-listes telles que tous les éléments de la première soient inférieurs à tous les éléments de la seconde, puis on trie les deux sous-listes selon le même processus jusqu'à avoir des sous-listes réduites à un seul élément.*

    (b) *on trie successivement les premiers éléments de la liste : à la $i^{\text{ème}}$ étape, on place le $i^{\text{ème}}$ élément à son rang parmi les $i-1$ éléments précédents qui sont déjà triés entre eux.*

    (c) *on parcourt la liste en commençant par la fin, en effectuant un échange à chaque fois que l'on trouve deux éléments successifs qui ne sont pas dans le bon ordre.*

    (d) *on recherche le minimum de la liste à trier, on le met en début de liste en l'échangeant avec le premier élément et on recommence sur le reste de la liste.*

Remarque 4.14 : *Parmi les 4 items de la question ci-contre, un seul item définit le tri par insertion, les 3 autres définissent d'autres méthodes de tri. Lesquelles ?*

4.5.2 Comprendre

TD 4.21 : Génération de séquences

A l'aide de la fonction `randint(min,max)` *du module standard* `random`*, Définir les fonctions de génération suivantes :*

1. `liste(n)` *: génère une liste de* `n` *entiers compris entre* `0` *et* `n`*.*
2. `nuplet(n)` *: génère un n-uplet de* `n` *entiers compris entre* `0` *et* `n`*.*
3. `chaine(n)` *: génère une chaîne de* `n` *caractères imprimables.*

TD 4.22 : Application d'une fonction à tous les éléments d'une liste

Définir une fonction qui applique la même fonction `f` *à tous les éléments d'une liste* `s`*.*

Exemples : `s = [-1,2,3,-4]` , `f = abs` → `s = [1,2,3,4]`
`s = [pi/2,pi,3*pi/2]` , `f = sin` → `s = [1,0,-1]`

TD 4.23 : Que fait cette procédure ?

On considère la procédure `f` *ci-dessous.*

```
def f(t,debut,fin):
    m = (debut + fin)/2
    while m > 0:
        for i in range(m,fin+1):
            j = i - m
            while j >= debut:
                print(m,i,j,t)
                if t[j] > t[j+m]:
                    t[j],t[j+m] = t[j+m],t[j]
                    j = j - m
                else: j = debut-1
        m = m/2
    return
```

1. *Tracer l'exécution* à la main *de l'appel* `f([4,2,1,2,3,5],0,5)`.
2. *Que fait cette procédure ?*

4.5.3 Appliquer

TD 4.24 : Codes ASCII et chaînes de caractères

1. *Ecrire un algorithme qui fournit le tableau des codes ASCII (annexe 4.6.2 page 194) associé à une chaîne (exemple :* `'bon'` → `[98, 111, 110]`*).*
2. *Ecrire un algorithme qui donne la chaîne de caractères associée à un tableau de codes ASCII (exemple :* `[98, 111, 110]` → `'bon'`*).*

TD 4.25 : Opérations sur les matrices

1. *Définir une fonction qui teste si une liste* `m` *est une matrice.*
2. *Définir une fonction qui teste si une matrice* `m` *est une matrice carrée.*
3. *Définir une fonction qui teste si une matrice* `m` *est une matrice symétrique.*
4. *Définir une fonction qui teste si une matrice* `m` *est une matrice diagonale.*
5. *Définir une fonction qui multiplie une matrice* `m` *par un scalaire* `x`*.*
6. *Définir une fonction qui détermine la transposée d'une matrice* `m`*.*

4.5.4 Analyser

TD 4.26 : Recherche d'un motif
Définir un algorithme qui recherche la première occurence d'une séquence `m` *au sein d'une autre séquence* `t`.

TD 4.27 : Recherche de toutes les occurences
Définir une fonction qui retourne la liste des rangs de toutes les occurences d'un élément `x` *dans une liste* `t`.

TD 4.28 : Tri bulles
Dans le tri bulles, on parcourt la liste en commençant par la fin, en effectuant un échange à chaque fois que l'on trouve deux éléments successifs qui ne sont pas dans le bon ordre.
Définir une fonction qui trie une liste selon la méthode du tri bulles.

TD 4.29 : Méthode d'élimination de Gauss
L'objectif est ici de résoudre dans $\mathbb{R}$ *un système de* n *équations linéaires à* n *inconnues, homogènes ou non homogènes, du type* $A \cdot x = b$ *:*

$$\left\{\begin{array}{rrrrrrr} a_{00}x_0 & + & a_{01}x_1 & + \cdots + & a_{0(n-1)}x_{(n-1)} & = & b_0 \\ a_{10}x_0 & + & a_{11}x_1 & + \cdots + & a_{1(n-1)}x_{(n-1)} & = & b_1 \\ \cdots & + & \cdots & + \cdots + & \cdots & = & \cdots \\ a_{(n-1)0}x_0 & + & a_{(n-1)1}x_1 & + \cdots + & a_{(n-1)(n-1)}x_{(n-1)} & = & b_{(n-1)} \end{array}\right.$$

Définir une fonction `solve(a,b)` *qui retourne le vecteur* `x` *solution du système linéaire* $A \cdot x = b$ *selon la méthode d'élimination de* Gauss *décrite en section 4.6.5 page 199.*

4.5.5 Evaluer

TD 4.30 : Comparaison d'algorithmes de recherche.
Comparer les temps d'exécution des versions récursive et itérative des 2 méthodes de recherche développées dans le cours (section 4.3).

On pourra utiliser la fonction `random()` *ou* `randint(min,max)` *du module standard* `random` *pour générer des listes de nombres aléatoires, et la fonction* `time()` *du module standard* `time` *pour mesurer le temps avant et après l'appel des fonctions.*

TD 4.31 : Comparaison d'algorithmes de tri
Comparer les temps d'exécution des différentes versions récursives et itératives des 3 méthodes de tri développées dans le cours (section 4.4).

On pourra utiliser la fonction `random()` *ou* `randint(min,max)` *du module standard* `random` *pour générer des listes de nombres aléatoires, et la fonction* `time()` *du module standard* `time` *pour mesurer le temps avant et après l'appel des fonctions.*

4.5.6 Solutions des exercices

TD 4.20 : QCM (4).

Les bonnes réponses sont extraites directement du texte de ce chapitre :
1c, 2d, 3d, 4c, 5b, 6d, 7c, 8a, 9c, 10b

TD 4.21 : Génération de séquences.

1. Génération de listes d'entiers.

```
def liste(n):
    assert type(n) is int and n >= 0
    t = []
    for i in range(n):
        t.append(randint(0,n))
    return t
```

```
>>> liste(0)
[]
>>> liste(10)
[6, 7, 9, 8, 5, 4, 9, 1, 8, 9]
```

2. Génération de n-uplets d'entiers.

```
def nuplet(n):
    assert type(n) is int and n >= 0
    t = ()
    for i in range(n):
        t = t + (randint(0,n),)
    return t
```

```
>>> nuplet(0)
()
>>> nuplet(10)
(6, 2, 3, 10, 9, 3, 4, 1, 3, 4)
```

3. Génération de chaînes de caractères.

```
def chaine(n):
    assert type(n) is int and n >= 0
    t = ''
    for i in range(n):
        t = t + chr(randint(32,127)
    return t
```

```
>>> chaine(0)
''
>>> chaine(10)
'kN K,-`Phe'
```

TD 4.22 : Application d'une fonction à tous les éléments d'une liste.

```
def application(f,t):
    assert type(t) is list
    for i in range(len(t)): t[i] = f(t[i])
    return t
```

```
s = [pi/2,pi,3*pi/2]
>>> application(sin,s)
[1.0, 1.2246063538223773e-16, -1.0]
```

La fonction prédéfinie `map(f,t)` fait la même chose.

```
>>> s = [pi/2,pi,3*pi/2]
>>> map(sin,s)
[1.0, 1.2246063538223773e-16, -1.0]
```

TD 4.23 : Que fait cette procédure ?

```
>>> f([4,2,1,2,3,5],0,5)
2 2 0 [4, 2, 1, 2, 3, 5]
2 3 1 [1, 2, 4, 2, 3, 5]
2 4 2 [1, 2, 4, 2, 3, 5]
2 4 0 [1, 2, 3, 2, 4, 5]
2 5 3 [1, 2, 3, 2, 4, 5]
1 1 0 [1, 2, 3, 2, 4, 5]
1 2 1 [1, 2, 3, 2, 4, 5]
1 3 2 [1, 2, 3, 2, 4, 5]
1 3 1 [1, 2, 2, 3, 4, 5]
1 4 3 [1, 2, 2, 3, 4, 5]
1 5 4 [1, 2, 2, 3, 4, 5]
```

Cette procédure trie une liste selon la méthode du tri par incrément décroissant (tri shell). Au premier passage, on considère des sous-listes d'éléments distants de `m = (debut+fin)/2` que l'on trie séparément. Au deuxième passage, on considère des sous-listes d'éléments distants de `m/2` que l'on trie séparément, et ainsi de suite. A chaque passage, l'incrément `m` est divisé par 2 et toutes les sous-listes sont triées. Le tri s'arrête quand l'incrément est nul.

TD 4.24 : Codes ASCII et chaînes de caractères.

1. Chaîne → codes ASCII

```
def codeASCII(s):
    code = []
    for i in range(len(s)):
        code.append(ord(s[i]))
    return code
```

```
>>> codeASCII('bon')
[98, 111, 110]
```

2. Codes ASCII → chaîne

```
def decodeASCII(code):
    s = ''
    for i in range(len(code)):
        s = s + chr(code[i])
    return s
```

```
>>> decodeASCII([98, 111, 110])
'bon'
```

TD 4.25 : Opérations sur les matrices.

1. Matrice.

```
def matrice(t):
    ok = True
    if type(t) is not list or t == []:
        ok = False
    elif type(t[0]) is not list:
        ok = False
    else:
        i, n, m = 1, len(t), len(t[0])
        while i < n and ok == True:
            if type(t[i]) is not list:
                ok = False
            elif len(t[i]) != m:
                ok = False
            i = i + 1
    return ok
```

```
>>> matrice(5)
False
>>> matrice([])
False
>>> matrice([5])
False
>>> matrice([[5]])
True
>>> matrice([[5,6],
             [7]])
False
>>> matrice([[5,6,7],
             [8,9]])
False
>>> matrice([[5,6,7],
             [8,9,3]])
True
```

2. Matrice carrée.

```
def matriceCarree(t):
    assert matrice(t)
    if len(t) > 0 and len(t[0]) == len(t):
        ok = True
    else: ok = False
    return ok
```

```
>>> matriceCarree([[4,5,6],
                   [7,8,9]])
False
>>> matriceCarree([[5,6],
                   [8,9]])
True
>>> matriceCarree([[]])
False
```

3. Matrice symétrique.

```
def matriceSymetrique(t):
    assert matriceCarree(t)
    ok,i = True,0
    while i < len(t) and ok == True:
        j = i + 1
        while j < len(t[0]) and ok == True:
            if t[i][j] != t[j][i]:
                ok = False
            else: j = j + 1
        i = i + 1
    return ok
```

```
>>> matriceSymetrique([[5,6],
                       [8,9]])
False
>>> matriceSymetrique([[5,6],
                       [6,9]])
True
>>> matriceSymetrique([[5,6,7],
                       [6,8,9],
                       [7,9,3]])
True
```

4. Matrice diagonale.

```
def matriceDiagonale(t):
    assert matriceCarree(t)
    ok,i = True,0
    while i < len(t) and ok == True:
        j = 0
        while j < len(t[0]) and ok == True:
            if i != j and t[i][j] != 0:
                ok = False
            else: j = j + 1
        i = i + 1
    return ok
```

```
>>> matriceDiagonale([[5,6],
                      [8,9]])
False
>>> matriceDiagonale([[5,0],
                      [0,9]])
True
>>> matriceDiagonale([[5,0,0],
                      [0,0,0],
                      [0,0,3]])
True
```

5. Multiplication d'une matrice par un scalaire.

```
def multiplicationScalaire(t,x):
    assert matrice(m)
    assert type(x) is int or type(x) is float
    for i in range(len(t)):
        for j in range(len(t[0])):
            t[i][j] = x*t[i][j]
    return
```

```
>>> multiplicationScalaire([[5,6],
                            [2,7]],3)
[[15, 18], [6, 21]]
```

6. Transposée d'une matrice.

```
def transposee(t):
    assert matrice(t)
    s = []
    for j in range(len(t[0])):
        s.append([])
        for i in range(len(t)):
            s[j].append(t[i][j])
    return s
```

```
>>> transposee([[1,2],[4,5]])
[[1, 4], [2, 5]]
>>> transposee([[1,2,3],[4,5,6]])
[[1, 4], [2, 5], [3, 6]]
```

TD 4.26 : Recherche d'un motif.

```
def motif(t,m,debut,fin):
    assert type(t) is list
    assert type(m) is list
    assert 0 <= debut <= fin < len(t)
    i,ok = debut,False
    while i + len(m) - 1 <= fin and not ok:
        if t[i:i+len(m)] == m and m != []:
            ok = True
        else: i = i + 1
    return ok,i
```

```
>>> motif([1,2,3,2,3,4],[2,4],0,5)
(False, 5)
>>> motif([1,2,3,2,3,4],[2,3],0,5)
(True, 1)
>>> motif([1,2,3,2,3,4],[2,3],2,5)
(True, 3)
>>> motif([1,2,3,2,3,4],[2,3,4],0,5)
(True, 3)
```

TD 4.27 : Recherche de toutes les occurences.

```
def rechercheTout(t,x):
    assert type(t) is list
    occurs = []
    for i in range(len(t)):
        if t[i] == x: occurs.append(i)
    return occurs
```

```
>>> rechercheTout([1,2,1,5,1],1)
[0, 2, 4]
>>> rechercheTout([1,2,1,5,1],2)
[1]
>>> rechercheTout([1,2,1,5,1],3)
[]
```

TD 4.28 : Tri bulles.

```
def triBulles(t,debut,fin):
    assert type(t) is list
    assert 0 <= debut <= fin < len(t)
    while debut < fin:
        for j in range(fin,debut,-1):
            if t[j] < t[j-1]:
                t[j],t[j-1] = t[j-1],t[j]
        debut = debut + 1
    return t
```

```
>>> s = [9,8,7,6,5,4]
>>> triBulles(s,0,len(s)-1)
[4, 5, 6, 7, 8, 9]
>>> s = [9,8,7,6,5,4]
>>> triBulles(s,1,4)
[9, 5, 6, 7, 8, 4]
```

TD 4.29 : Méthode d'élimination de Gauss.

```
def solve(a,b):
    assert matriceCarree(a)
    assert type(b) is list
    assert len(a) == len(b)
    if triangularisation(a,b) == True:
        x = backsubstitutions(a,b)
    else: x = []
    return x

>>> a, b = [[4]], [1]
>>> solve(a,b)
[0.25]
>>> a, b = [[1,1],[1,-1]], [1,0]
>>> solve(a,b)
[0.5, 0.5]
>>> a = [[2,-1,2],[1,10,-3],[-1,2,1]]
>>> b = [2,5,-3]
>>> solve(a,b)
[2.0, 0.0, -1.0]
```

```
>>> a = [[ 1,  0, 0, -1,  1, 0],
         [ 1,  1, 0,  0,  0,-1],
         [ 0,  1,-1,  0, -1, 0],
         [10,-10, 0,  0,-10, 0],
         [ 0,  0, 5,-20,-10, 0],
         [ 0, 10, 5,  0,  0,10]]
>>> b = [0,0,0,0, 0,12]
>>> solve(a,b)
[0.25945945945945936,
 0.35675675675675667,
 0.45405405405405402,
 0.16216216216216217,
-0.097297297297297303,
 0.61621621621621636]
```

```
>>> a, b = [[10,7,8,7],[7,5,6,5],[8,6,10,9],[7,5,9,10]], [32,23,33,31]
>>> solve(a,b)
[1.000000000000102, 0.99999999999998335, 1.000000000000004, 0.99999999999999778]
>>> a = [[10,7,8.1,7.2],[7.08,5.04,6,5],[8,5.98,9.89,9],[6.99,4.99,9,9.98]]
>>> solve(a,b)
[28714.563366527444, -47681.976061485781, 12326.759748689999, -7387.0215554531824]
>>> a = [[10,7,8,7],[7,5,6,5],[8,6,10,9],[7,5,9,10]], [32.01,22.99,33.01,30.99]
>>> solve(a,b)
[1.8199999999999306, -0.35999999999988425, 1.3499999999999692, 0.79000000000001847]
```

```
def pivot(a,i0):
    maxi = fabs(a[i0][i0])
    r = i0
    for i in range(i0+1,len(a)):
      if fabs(a[i][i0]) > maxi:
          maxi = fabs(a[i][i0])
          r = i
    return r

def triangularisation(a,b):
   ok = True; i = 0
   while i < len(a) and ok == True:
      p = pivot(a,i)
      if i != p:
         a[i],a[p] = a[p],a[i]
         b[i],b[p] = b[p],b[i]
      if a[i][i] == 0: ok = False
      else:
         for k in range(i+1,len(a)):
            substractRows(a,b,k,i)
      i = i + 1
   return ok
```

```
def substractRows(a,b,k,i):
    q = 1.*a[k][i]/a[i][i]
    a[k][i] = 0
    b[k] = b[k] - q*b[i]
    for j in range(i+1,len(a)):
        a[k][j] = a[k][j] - q*a[i][j]
    return

def backsubstitutions(a,b):
    n = len(a); x = []
    for k in range(n): x.append(0)

    x[n-1] = 1.*b[n-1]/a[n-1][n-1]
    for i in range(n-2,-1,-1):
        x[i] = b[i]
        for j in range(i+1,n):
            x[i] = x[i] - a[i][j]*x[j]
        x[i] = 1.*x[i]/a[i][i]
    return x
```

TD 4.30 : Comparaison d'algorithmes de recherche.

```
def mesureRecherche(f,t,x):
    t0 = time()
    f(tmp,x,0,len(t)-1)
    dt = time() - t0
    return dt
```

```
>>> s = liste(1000)
>>> x = s[len(s)-1]
>>> mesureRecherche(rechercheSequentielle,s,x)
0.00062489509582519531
>>> s = liste(100000)
>>> x = s[len(s)-1]
>>> mesureRecherche(rechercheSequentielle,s,x)
0.046545028686523438
```

TD 4.31 : Comparaison d'algorithmes de tri.

```
def mesureTri(f,t):
    tmp = [x for x in t]
    t0 = time()
    f(tmp,0,len(t)-1)
    dt = time() - t0
    return dt
```

```
>>> s = liste(10000)
>>> mesureTri(triSelection,s)
23.040315866470337
>>> mesureTri(triInsertion,s)
22.086866855621338
>>> mesureTri(triRapide,s)
0.24324798583984375
```

4.6 Annexes

4.6.1 Type abstrait de données

Nous nous intéressons ici à une description plus formelle du type abstrait de données `Sequence` selon un formalisme logique inspiré de [13]. On notera $\mathbb{B}$ l'ensemble des booléens ($\{0,1\}$), $\mathbb{E}$ l'ensemble des éléments, $\mathbb{S}$ l'ensemble des séquences, $\mathbb{F}$ l'ensemble des fonctions $\mathbb{E} \rightarrow \mathbb{E}$ et $\mathbb{P}$ l'ensemble des fonctions $\mathbb{E} \rightarrow \mathbb{B}$. Le vocabulaire initial de la théorie des séquences comprend alors :

- une constante `[]` qui représente la séquence vide (on notera $\mathbb{S}^* = \mathbb{S} - \{\texttt{[]}\}$) ;
- un opérateur $x \circ s$ qui traduit l'insertion d'un élément $x \in \mathbb{E}$ en tête de la séquence $s \in \mathbb{S}$.

Dans ce cadre, la définition axiomatique des séquences repose sur les 2 axiomes suivants :

1. `[]` est une séquence : `[]` $\in \mathbb{S}$,
2. $x \circ s$ est une séquence : $\forall x \in \mathbb{E},\ \forall s \in \mathbb{S},\ x \circ s \in \mathbb{S}$.

Pour chaque opération considérée, nous indiquerons sa signature, sa définition axiomatique (souvent récursive) et son équivalent Python.

- head(s) : tête (premier élément) d'une séquence
  head $: \mathbb{S}^* \rightarrow \mathbb{E}$, $x = \text{head}(x \circ s)$ `s[0]`
- tail(s) : queue d'une séquence (la séquence sans son premier élément)
  tail $: \mathbb{S}^* \rightarrow \mathbb{E}$, $s = \text{tail}(x \circ s)$ `s[1:len(s)]`
- x in s : appartenance d'un élément à une séquence
  in $: \mathbb{E} \times \mathbb{S} \rightarrow \mathbb{B}$, $\left| \begin{array}{l} x = \text{head}(s) \\ x \text{ in tail}(s) \end{array} \right.$ `x in s`
- len(s) : longueur d'une séquence (nombre d'éléments)
  len $: \mathbb{S} \rightarrow \mathbb{N}$, $\left| \begin{array}{l} \text{len}(\texttt{[]}) = 0 \\ \text{len}(x \circ s) = \text{len}(s) + 1 \end{array} \right.$ `len(s)`
- concat(s_1, s_2) (ou $s_1 + s_2$) : concaténation de deux séquences
  concat $: \mathbb{S} \times \mathbb{S} \rightarrow \mathbb{S}$, $\left| \begin{array}{l} \texttt{[]} + s = s \\ s_3 = (x \circ s_1) + s_2 = x \circ (s_1 + s_2) \end{array} \right.$ `s1 + s2`
- ith(s, i) (ou s_i) : $i^{ème}$ élément de la séquence
  ith $: \mathbb{S}^* \times \mathbb{N} \rightarrow \mathbb{E}$, $\left| \begin{array}{l} (x \circ s)_0 = x \\ (x \circ s)_i = s_{i-1} \quad 0 < i \leq \text{len}(s) \end{array} \right.$ `s[i]`

- foreach(f, s) : application d'une fonction à tous les éléments d'une séquence

$$\text{foreach} : \mathbb{F} \times \mathbb{S} \rightarrow \mathbb{S} \quad , \left| \begin{array}{l} \text{foreach}(f, \texttt{[]}) = \texttt{[]} \\ \text{foreach}(f, x \circ s) = f(x) \circ \text{foreach}(f, s) \end{array} \right.$$

`[f(x) for x in s]`

- collect(p, s) : sélection sous condition des éléments d'une séquence

$$\text{collect} : \mathbb{P} \times \mathbb{S} \rightarrow \mathbb{S} \quad , \left| \begin{array}{l} \text{collect}(p, \texttt{[]}) = \texttt{[]} \\ \text{collect}(p, x \circ s) = x \circ \text{collect}(p, s) \text{ si } p(x) \\ \text{collect}(p, x \circ s) = \text{collect}(p, s) \text{ si } \neg p(x) \end{array} \right.$$

`[x for x in s if p(x)]`

4.6.2 Codes ASCII

L'ordinateur stocke toutes les données sous forme numérique (ensemble de bits). En particulier, les caractères ont un équivalent numérique : le code ASCII (*American Standard Code for Information Interchange*). Le code ASCII de base code les caractères sur 7 bits (128 caractères, de 0 à 127).

- Les codes 0 à 31 sont des caractères de contrôle (figure 4.19).
- Les codes 32 à 47, de 58 à 64, de 91 à 96 et de 123 à 126 sont des symboles de ponctuation.

32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
	!	"	#	$	%	&	’	(	)	*	+	,	-	.	/

58	59	60	61	62	63	64
:	;	<	=	>	?	@

91	92	93	94	95	96
[	\	]	^	_	‘

123	124	125	126
{	|	}	~

- Les codes de 48 à 57 représentent les 10 chiffres de 0 à 9.
- Les codes 65 à 90 représentent les majuscules de `A` à `Z`,
- Les codes 97 à 122 représentent les minuscules de `a` à `z` (il suffit d'ajouter 32 au code ASCII d'une majuscule pour obtenir la minuscule correspondante).
- Les caractères accentués (`é`, `è` ...) font l'objet d'un code ASCII étendu de 128 à 255.

4.6.3 Les séquences en Python

Les principales opérations sur les séquences en Python (`list`, `tuple`, `str`) sont listées dans les tableaux ci-dessous, extraits du Python *2.5 Quick Reference Guide* [10].

Fig. 4.19 : Codes des caractères de contrôle

code	*caractère*	*signification*
0	*NUL*	Null
1	*SOH*	Start of heading
2	*STX*	Start of text
3	*ETX*	End of text
4	*EOT*	End of transmission
5	*ENQ*	Enquiry
6	*ACK*	Acknowledge
7	*BEL*	Bell
8	*BS*	Backspace
9	*TAB*	Horizontal tabulation
10	*LF*	Line Feed
11	*VT*	Vertical tabulation
12	*FF*	Form feed
13	*CR*	Carriage return
14	*SO*	Shift out
15	*SI*	Shift in
16	*DLE*	Data link escape
17	*DC1*	Device control 1
18	*DC2*	Device control 2
19	*DC3*	Device control 3
20	*DC4*	Device control 4
21	*NAK*	Negative acknowledgement
22	*SYN*	Synchronous idle
23	*ETB*	End of transmission block
24	*CAN*	Cancel
25	*EM*	End of medium
26	*SUB*	Substitute
27	*ESC*	Escape
28	*FS*	File separator
29	*GS*	Group separator
30	*RS*	Record separator
31	*US*	Unit separator

Operation on sequences (list, tuple, str)	**Result**
x in s	True if an item of s is equal to x, else False
x not in s	False if an item of s is equal to x, else True
s1 + s2	the concatenation of s1 and s2
s * n, n*s	n copies of s concatenated
s[i]	i'th item of s, origin 0
s[i :j[:step]]	Slice of s from i (included) to j(excluded). Optional step value, possibly negative (default : 1)
len(s)	Length of s
min(s)	Smallest item of s
max(s)	Largest item of s

Operation on list	**Result**
s[i] = x	item i of s is replaced by x
s[i :j [:step]] = t	slice of s from i to j is replaced by t
del s[i :j[:step]]	same as s[i :j] = []
s.count(x)	returns number of i's for which s[i] == x
s.index(x[,start[,stop]])	returns smallest i such that s[i] == x. start and stop limit search to only part of the list
s.append(x)	same as s[len(s) : len(s)] = [x]
s.extend(x)	same as s[len(s) :len(s)]= x
s.insert(i, x)	same as s[i :i] = [x] if i>= 0. i == -1 inserts before the last element
s.remove(x)	same as del s[s.index(x)]
s.pop([i])	same as x = s[i] ; del s[i] ; return x
s.reverse()	reverses the items of s in place
s.sort([cmp])	sorts the items of s in place

Operation on `str`	**Result**
`s.capitalize()`	Returns a copy of `s` with its first character capitalized, and the rest of the characters lowercased
`s.center(width[, fillChar=' '])`	Returns a copy of `s` centered in a string of length `width`, surrounded by the appropriate number of `fillChar` characters
`s.count(sub[,start[, end]])`	Returns the number of occurrences of substring `sub` in string `s`
`s.decode([encoding[, errors]])`	Returns a unicode string representing the decoded version of str `s`, using the given codec (`encoding`). Useful when reading from a file or a I/O function that handles only str. Inverse of `encode`
`s.encode([encoding[, errors]])`	Returns a str representing an encoded version of `s`. Mostly used to encode a unicode string to a str in order to print it or write it to a file (since these I/O functions only accept str). Inverse of `decode`
`s.endswith(suffix[, start[,end]])`	Returns `True` if `s` ends with the specified `suffix`, otherwise return `False`.
`s.expandtabs([tabsize])`	Returns a copy of `s` where all tab characters are expanded using spaces
`s.find(sub[,start[, end]])`	Returns the lowest index in `s` where substring `sub` is found. Returns `-1` if `sub` is not found
`s.index(sub[,start[, end]])`	like `find()`, but raises `ValueError` when the substring `sub` is not found
`s.isalnum()`	Returns `True` if all characters in `s` are alphanumeric, `False` otherwise
`s.isalpha()`	Returns `True` if all characters in `s` are alphabetic, `False` otherwise
`s.isdigit()`	Returns `True` if all characters in `s` are digit characters, `False` otherwise
`s.isspace()`	Returns `True` if all characters in `s` are whitespace characters, `False` otherwise

`s.istitle()`	Returns `True` if string `s` is a titlecased string, `False` otherwise
`s.islower()`	Returns `True` if all characters in `s` are lowercase,`False` otherwise
`s.isupper()`	Returns `True` if all characters in `s` are uppercase, `False` otherwise
`separator.join(seq)`	Returns a concatenation of the strings in the sequence `seq`, separated by string `separator`, e.g. : `",".join(['A','B','C']) -> "A,B,C"`
`s.ljust/rjust/center( width[,fillChar=' '])`	Returns `s` left/right justified/centered in a string of length `width`
`s.lower()`	Returns a copy of `s` converted to lowercase
`s.lstrip([chars])`	Returns a copy of `s` with leading `chars` (default : blank chars) removed
`s.partition(separ)`	Searches for the separator `separ` in `s`, and returns a tuple `(head, sep, tail)` containing the part before it, the separator itself, and the part after it. If the separator is not found, returns `s` and two empty strings
`s.replace(old,new[, maxCount=-1])`	Returns a copy of `s` with the first `maxCount` (`-1` : unlimited) occurrences of substring `old` replaced by `new`
`s.rfind(sub[ ,start[, end]])`	Returns the highest index in `s` where substring `sub` is found. Returns `-1` if `sub` is not found
`s.rindex(sub[,start[, end]])`	like `rfind()`, but raises `ValueError` when the substring `sub` is not found
`s.rpartition(separ)`	Searches for the separator `separ` in `s`, starting at the `end` of `s`, and returns a tuple `(tail, sep, head)` containing the part before it, the separator itself, and the part after it. If the separator is not found, returns two empty strings and `s`
`s.rstrip([chars])`	Returns a copy of `s` with trailing `chars` (default : blank chars) removed, e.g. `aPath.rstrip('/')` will remove the trailing `'/'` from `aPath` if it exists

`s.split([separator[, maxsplit]])`	Returns a list of the words in `s`, using `separator` as the delimiter string
`s.rsplit([separator[, maxsplit]])`	Same as `split`, but splits from the end of the string
`s.splitlines([keepends])`	Returns a list of the lines in `s`, breaking at line boundaries
`s.startswith(prefix[, start[,end]])`	Returns `True` if `s` starts with the specified `prefix`, otherwise returns `False`. Negative numbers may be used for `start` and `end`
`s.strip([chars])`	Returns a copy of `s` with leading and trailing `chars` (default : blank chars) removed
`s.swapcase()`	Returns a copy of `s` with uppercase characters converted to lowercase and vice versa
`s.title()`	Returns a titlecased copy of `s`, i.e. words start with uppercase characters, all remaining cased characters are lowercase
`s.translate(table[, deletechars])`	Returns a copy of `s` mapped through translation table `table`
`s.upper()`	Returns a copy of `s` converted to uppercase
`s.zfill(width)`	Returns the numeric string left filled with zeros in a string of length `width`

4.6.4 Les fichiers en Python

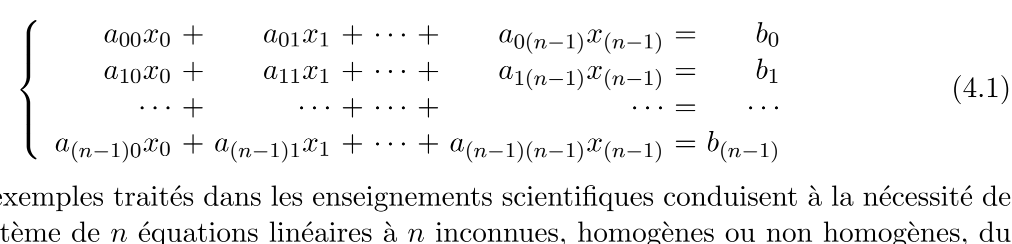
Les principales opérations sur les fichiers en Python (type `file`) sont listées dans le tableau ci-dessous, extrait du Python *2.5 Quick Reference Guide* [10].

`open(filename[,mode='r',[bufsize]])` returns a new file object. `filename` is the file name to be opened. `mode` indicates how the file is to be opened (`'r'`, `'w'`, `'a'`, `'+'`, `'b'`, `'U'`). `bufsize` is `0` for unbuffered, `1` for line buffered, negative or omitted for system default, `>1` for a buffer of (about) the given size.

Operation on `file`	**Result**
`f.close()`	Close file `f`
`f.fileno()`	Get fileno (`fd`) for file `f`
`f.flush()`	Flush file `f`'s internal buffer
`f.isatty()`	`True` if file `f` is connected to a tty-like dev, else `False`
`f.next()`	Returns the next input line of file `f`, or raises `StopIteration` when `EOF` is hit
`f.read([size])`	Read at most `size` bytes from file `f` and return as a string object. If `size` omitted, read to `EOF`
`f.readline()`	Read one entire line from file `f`. The returned line has a trailing `n`, except possibly at `EOF`. Return `''` on `EOF`
`f.readlines()`	Read until `EOF` with `readline()` and return a list of lines read
`for line in f : ...`	Iterate over the `lines` of a file `f` (using `readline`)
`f.seek(offset[, whence=0])`	Set file `f`'s position `whence == 0` then use absolute indexing `whence == 1` then `offset` relative to current pos `whence == 2` then `offset` relative to file end
`f.tell()`	Return file `f`'s current position (byte offset)
`f.truncate([size])`	Truncate `f`'s `size`. If `size` is present, `f` is truncated to (at most) that `size`, otherwise `f` is truncated at current position (which remains unchanged)
`f.write(str)`	Write string `str` to file `f`
`f.writelines(list)`	Write `list` of strings to file `f`. No `EOL` are added

4.6.5 Méthode d'élimination de Gauss

L'objectif est ici de résoudre dans $\mathbb{R}$ un système de n équations linéaires à n inconnues, homogènes ou non homogènes, du type $A \cdot x = b$:



$$\left\{\begin{array}{rrrr} a_{00}x_0 + & a_{01}x_1 + \cdots + & a_{0(n-1)}x_{(n-1)} = & b_0 \\ a_{10}x_0 + & a_{11}x_1 + \cdots + & a_{1(n-1)}x_{(n-1)} = & b_1 \\ \cdots + & \cdots + \cdots + & \cdots = & \cdots \\ a_{(n-1)0}x_0 + & a_{(n-1)1}x_1 + \cdots + & a_{(n-1)(n-1)}x_{(n-1)} = & b_{(n-1)} \end{array}\right. \quad (4.1)$$

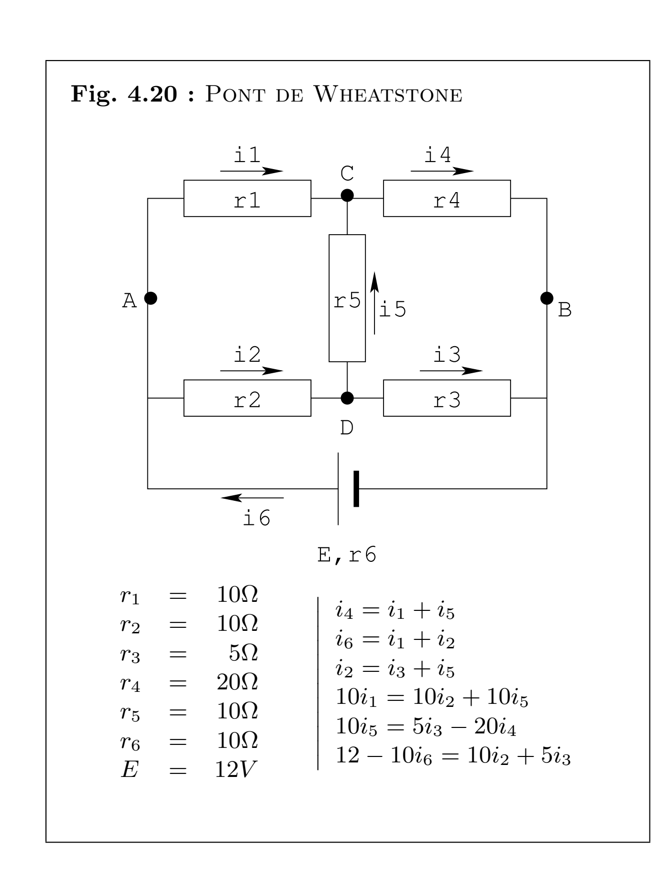


Fig. 4.20 : PONT DE WHEATSTONE

De nombreux exemples traités dans les enseignements scientifiques conduisent à la nécessité de résoudre un système de n équations linéaires à n inconnues, homogènes ou non homogènes, du type $A{\cdot}x = b$. La figure 4.20 propose à titre d'exemple le cas du pont de Wheatstone en électricité. Les ponts ont été utilisés pour la mesure des résistances, inductances et capacités jusqu'à ce que les progrès en électronique les rendent obsolètes en métrologie. Toutefois la structure en pont reste encore utilisée dans de nombreux montages (voir par exemple [14]).

La première méthode généralement utilisée pour trouver la solution d'un système d'équations linéaires tel que le système (4.1) est celle qui consiste à éliminer les inconnues (x_i) en combinant les équations [8]. Pour illustrer cette méthode, nous commencerons par étudier un exemple simple pouvant s'effectuer *à la main*, puis nous passerons au cas général pour présenter la méthode d'élimination de GAUSS.

Etude d'un cas particulier

Considérons le système suivant :

$$\left\{\begin{array}{rcrcrcr} x_0 & + & x_1 & + & x_2 & = & 1 \\ 2x_0 & + & 4x_1 & + & 8x_2 & = & 10 \\ 3x_0 & + & 9x_1 & + & 27x_2 & = & 33 \end{array}\right.$$

Pour résoudre un tel système, l'idée de base est de triangulariser le système de telle manière qu'il soit possible de remonter les solutions par substitutions successives.

La première étape consiste à éliminer le terme en x_0 des 2ème et 3ème équations. Cherchons tout d'abord à éliminer le terme en x_0 de la 2ème équation. Pour cela nous multiplions la 1ère équation par le coefficient de x_0 de la 2ème équation ($a_{10} = 2$). On obtient le nouveau système :

$$\left\{\begin{array}{rcrcrcr} 2x_0 & + & 2x_1 & + & 2x_2 & = & 2 \\ 2x_0 & + & 4x_1 & + & 8x_2 & = & 10 \\ 3x_0 & + & 9x_1 & + & 27x_2 & = & 33 \end{array}\right.$$

On soustrait alors la première équation de la deuxième équation, ce qui conduit au système :

$$\left\{\begin{array}{rcrcrcr} 2x_0 & + & 2x_1 & + & 2x_2 & = & 2 \\ & & 2x_1 & + & 6x_2 & = & 8 \\ 3x_0 & + & 9x_1 & + & 27x_2 & = & 33 \end{array}\right.$$

On recommence l'opération pour éliminer le terme en x_0 de la 3ème équation. Pour cela, on ramène à 1 le coefficient de x_0 dans la première équation en divisant cette équation par 2 ($a_{00} = 2$), puis nous multiplions la 1ère équation par le coefficient de x_0 de la 3ème équation ($a_{20} = 3$).

$$\left\{\begin{array}{rcrcrcr} 3x_0 & + & 3x_1 & + & 3x_2 & = & 3 \\ & & 2x_1 & + & 6x_2 & = & 8 \\ 3x_0 & + & 9x_1 & + & 27x_2 & = & 33 \end{array}\right.$$

On soustrait ensuite la 1ère équation de la 3ème équation :

$$\left\{\begin{array}{rcrcrcr} 3x_0 & + & 3x_1 & + & 3x_2 & = & 3 \\ & & 2x_1 & + & 6x_2 & = & 8 \\ & & 6x_1 & + & 24x_2 & = & 30 \end{array}\right.$$

On obtient un nouveau système linéaire dans lequel seule la première équation contient un terme en x_0. L'équation utilisée (ici la 1ère équation) pour éliminer une inconnue dans les équations qui suivent (ici les 2ème et 3ème équations) est appelée l'équation-pivot. Dans l'équation-pivot choisie, le coefficient de l'inconnue qui est éliminée dans les autres équations est appelé le pivot de l'équation (ici a_{00}).

La deuxième étape consiste à éliminer le terme en x_1 de la troisième équation en utilisant la deuxième équation comme équation-pivot. On ramène à 1 le coefficient de x_1 dans la 2ème équation en divisant l'équation par 2 ($a_{11} = 2$), puis on la multiplie par 6 ($a_{21} = 6$) pour que les 2ème et 3ème équations aient le même terme en x_1. Tout revient à multiplier la 2ème équation par 3 ($a_{21}/a_{11} = 6/2 = 3$) :

$$\left\{\begin{array}{rcrcrcr} 3x_0 & + & 3x_1 & + & 3x_2 & = & 3 \\ & & 6x_1 & + & 18x_2 & = & 24 \\ & & 6x_1 & + & 24x_2 & = & 30 \end{array}\right.$$

Il reste à soustraire la 2ème équation de la 3ème pour éliminer le terme en x_1 de la 3ème équation :

$$\left\{\begin{array}{rcrcrcr} 3x_0 & + & 3x_1 & + & 3x_2 & = & 3 \\ & & 6x_1 & + & 18x_2 & = & 24 \\ & & & & 6x_2 & = & 6 \end{array}\right.$$

On obtient ainsi un système triangulaire d'équations linéaires dont on peut calculer directement la valeur de x_2 par la 3ème équation : $6x_2 = 6 \Rightarrow x_2 = 1$. On porte cette valeur de x_2 dans la deuxième équation, ce qui nous permet de calculer x_1 : $6x_1 + 18 \cdot 1 = 24 \Rightarrow x_1 = 1$. En reportant les valeurs de x_2 et x_1 dans la première équation, on en déduit la valeur de x_0 : $3x_0 + 3 \cdot 1 + 3 \cdot 1 = 3 \Rightarrow x_0 = -1$. On vérifie simplement que les valeurs obtenues sont solutions du système initial :

$$\left\{\begin{array}{rcrcrcr} x_0 & + & x_1 & + & x_2 & = & 1 \\ 2x_0 & + & 4x_1 & + & 8x_2 & = & 10 \\ 3x_0 & + & 9x_1 & + & 27x_2 & = & 33 \end{array}\right. \qquad \begin{pmatrix} 1 & 1 & 1 \\ 2 & 4 & 8 \\ 3 & 9 & 27 \end{pmatrix} \cdot \begin{pmatrix} -1 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 10 \\ 33 \end{pmatrix}$$

Etude du cas général

De manière générale, la méthode précédente consiste à réduire le système de n équations à n inconnues à un système triangulaire équivalent qui peut être ensuite résolu facilement par substitutions. En quelque sorte le système (4.1) doit être transformé en un système équivalent du type :

$$\left\{\begin{array}{rcrcrcccrcl} a_{00}x_0 & + & a_{01}x_1 & + & a_{02}x_2 & + & \cdots & + & a_{0(n-1)}x_{(n-1)} & = & b_0 \\ & & a_{11}^{(1)}x_1 & + & a_{12}^{(1)}x_2 & + & \cdots & + & a_{1(n-1)}^{(1)}x_{(n-1)} & = & b_1^{(1)} \\ & & & + & a_{22}^{(2)}x_2 & + & \cdots & + & a_{2(n-1)}^{(2)}x_{(n-1)} & = & b_2^{(2)} \\ & & & & & & \cdots & + & \cdots & = & \cdots \\ & & & & & & & & a_{(n-1)(n-1)}^{(n-2)}x_{(n-1)} & = & b_{(n-1)}^{(n-2)} \end{array}\right. \tag{4.2}$$

où l'indice supérieur désigne le nombre d'étapes à la suite desquelles est obtenu le coefficient considéré.

L'équation de rang 0 du système (4.1) est d'abord divisée par le coefficient a_{00} de x_0 (supposé non nul). On obtient :

$$x_0 + \frac{a_{01}}{a_{00}}x_1 + \frac{a_{02}}{a_{00}}x_2 + \cdots + \frac{a_{0(n-1)}}{a_{00}}x_{(n-1)} = \frac{b_0}{a_{00}} \tag{4.3}$$

Cette équation (4.3) est alors multiplié par a_{10}, coefficient de x_0 dans la deuxième équation du système (4.1). Le résultat est ensuite soustrait de la deuxième équation du système (4.1), ce qui élimine x_0 dans la deuxième équation. D'une manière générale on multiplie la relation (4.3) par a_{i0}, coefficient de x_0 dans l'équation de rang i du système (4.1) et on retranche le résultat obtenu de cette même $i^{ème}$ équation. A la fin x_0 est éliminé de toutes les équations, excepté de la première, et on obtient le système ainsi transformé :

$$\left\{\begin{array}{rcrcrcccrcl} a_{00}x_0 & + & a_{01}x_1 & + & a_{02}x_2 & + & \cdots & + & a_{0(n-1)}x_{(n-1)} & = & b_0 \\ & & a_{11}^{(1)}x_1 & + & a_{12}^{(1)}x_2 & + & \cdots & + & a_{1(n-1)}^{(1)}x_{(n-1)} & = & b_1^{(1)} \\ & & a_{21}^{(1)}x_1 & + & a_{22}^{(1)}x_2 & + & \cdots & + & a_{2(n-1)}^{(1)}x_{(n-1)} & = & b_2^{(1)} \\ & & a_{31}^{(1)}x_1 & + & a_{32}^{(1)}x_2 & + & \cdots & + & a_{3(n-1)}^{(1)}x_{(n-1)} & = & b_3^{(1)} \\ & & \cdots & + & \cdots & + & \cdots & + & \cdots & = & \cdots \\ & & a_{(n-1)1}^{(1)}x_1 & + & a_{(n-1)2}^{(1)}x_2 & + & \cdots & + & a_{(n-1)(n-1)}^{(1)}x_{(n-1)} & = & b_{(n-1)}^{(1)} \end{array}\right. \tag{4.4}$$

L'équation de rang 1 dans le nouveau système (4.4) devient alors l'équation-pivot et $a_{11}^{(1)}$ le pivot de l'équation. De la même façon on élimine x_1 des équations du rang 2 au rang $n-1$ dans le système (4.4), et on obtient :

$$\left\{\begin{array}{rcrcrcccrcl} a_{00}x_0 & + & a_{01}x_1 & + & a_{02}x_2 & + & \cdots & + & a_{0(n-1)}x_{(n-1)} & = & b_0 \\ & & a_{11}^{(1)}x_1 & + & a_{12}^{(1)}x_2 & + & \cdots & + & a_{1(n-1)}^{(1)}x_{(n-1)} & = & b_1^{(1)} \\ & & & & a_{22}^{(2)}x_2 & + & \cdots & + & a_{2(n-1)}^{(2)}x_{(n-1)} & = & b_2^{(2)} \\ & & & & a_{32}^{(2)}x_2 & + & \cdots & + & a_{3(n-1)}^{(2)}x_{(n-1)} & = & b_3^{(2)} \\ & & & & \cdots & + & \cdots & + & \cdots & = & \cdots \\ & & & & a_{(n-1)2}^{(2)}x_2 & + & \cdots & + & a_{(n-1)(n-1)}^{(2)}x_{(n-1)} & = & b_{(n-1)}^{(2)} \end{array}\right. \tag{4.5}$$

L'équation de rang 2 dans le nouveau système (4.5) fait office à son tour d'équation pivot, et ainsi de suite jusqu'à obtenir le système triangulaire (4.2). Une fois obtenu ce système triangulaire, on calcule directement la valeur de $x_{(n-1)}$ par la dernière relation du système triangulaire. En portant cette valeur dans la relation précédente, on calculera $x_{(n-2)}$, et ainsi de suite en remontant le système triangulaire (4.2). A chaque étape k, on a donc les relations suivantes :

1. Triangularisation

$$\left\{\begin{array}{lcl} a_{ij}^{(k)} & = & a_{ij}^{(k-1)} - \dfrac{a_{ip}^{(k-1)}}{a_{pp}^{(k-1)}} \cdot a_{pj}^{(k-1)} \\ b_{i}^{(k)} & = & b_{i}^{(k-1)} - \dfrac{a_{ip}^{(k-1)}}{a_{pp}^{(k-1)}} \cdot b_{p}^{(k-1)} \end{array}\right. \quad \text{avec} \left\{\begin{array}{l} k \ : \ \text{étape} \\ p \ : \ \text{rang du pivot} \\ \\ p < i \leq (n-1) \\ p \leq j \leq (n-1) \end{array}\right.$$

2. Remontée par substitutions

$$\left\{\begin{array}{lcl} x_{(n-1)} & = & \dfrac{b_{(n-1)}}{a_{(n-1)(n-1)}} \\ x_i & = & \dfrac{b_i - \displaystyle\sum_{j=i+1}^{n-1} a_{ij}x_j}{a_{ii}} \end{array}\right. \quad \text{avec } 0 \leq i < (n-1)$$

Remarque 4.15 : *Cette méthode est connue sous le nom de méthode d'élimination de* GAUSS *(également connue sous le nom de méthode de triangularisation de* GAUSS *ou méthode du pivot de* GAUSS*). Elle fut nommée ainsi en l'honneur du mathématicien allemand* JOHANN CARL FRIEDRICH GAUSS *(1777–1855), mais elle est connue des Chinois depuis au moins le* 1er *siècle de notre ère. Elle est référencée dans le livre chinois « Jiuzhang suanshu » où elle est attribuée à* CHANG TS'ANG *chancelier de l'empereur de Chine au* 2ème *siècle avant notre ère [4].*

Jusqu'à présent, on a admis que le pivot au cours du processus de triangularisation était non nul ($a_{pp} \neq 0$). Si ce n'est pas le cas, on doit permuter l'équation-pivot avec une autre ligne dans laquelle le pivot est différent de 0. Il se peut également que le pivot, sans être nul, soit très petit et l'on a intérêt là-aussi à interchanger les lignes comme pour le cas d'un pivot nul. En fait, pour augmenter la précision de la solution obtenue, on a toujours intérêt à utiliser la ligne qui a le plus grand pivot.

Jeu de tests

L'algorithme de résolution de tels systèmes linéaires pourra être testé à l'aide des systèmes suivants :

Test	**Système** $A \cdot x = b$		**Solution exacte**
	A	b	
1	$\begin{pmatrix} 4 \end{pmatrix}$	$\begin{pmatrix} 1 \end{pmatrix}$	$\begin{pmatrix} 1/4 \end{pmatrix}$
2	$\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$	$\begin{pmatrix} 1 \\ 0 \end{pmatrix}$	$\begin{pmatrix} 1/2 \\ 1/2 \end{pmatrix}$
3	$\begin{pmatrix} 2 & -1 & 2 \\ 1 & 10 & -3 \\ -1 & 2 & 1 \end{pmatrix}$	$\begin{pmatrix} 2 \\ 5 \\ -3 \end{pmatrix}$	$\begin{pmatrix} 2 \\ 0 \\ -1 \end{pmatrix}$

Test	**Système $A \cdot x = b$**		**Solution exacte**
	A	b	
4	$\begin{pmatrix} 10 & 7 & 8 & 7 \\ 7 & 5 & 6 & 5 \\ 8 & 6 & 10 & 9 \\ 7 & 5 & 9 & 10 \end{pmatrix}$	$\begin{pmatrix} 32 \\ 23 \\ 33 \\ 31 \end{pmatrix}$	$\begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \end{pmatrix}$
5	$\begin{pmatrix} 10 & 7 & 8.1 & 7.2 \\ 7.08 & 5.04 & 6 & 5 \\ 8 & 5.98 & 9.89 & 9 \\ 6.99 & 4.99 & 9 & 9.98 \end{pmatrix}$	$\begin{pmatrix} 32 \\ 23 \\ 33 \\ 31 \end{pmatrix}$	—
6	$\begin{pmatrix} 10 & 7 & 8 & 7 \\ 7 & 5 & 6 & 5 \\ 8 & 6 & 10 & 9 \\ 7 & 5 & 9 & 10 \end{pmatrix}$	$\begin{pmatrix} 32.01 \\ 22.99 \\ 33.01 \\ 30.99 \end{pmatrix}$	—

Test	**Système $A \cdot x = b$**	
	A	b
7	$\begin{pmatrix} 1 & 0 & 0 & - & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 & -1 \\ 0 & 1 & -1 & 0 & -1 & 0 \\ 10 & -10 & 0 & 0 & -10 & 0 \\ 0 & 0 & 5 & -20 & -10 & 0 \\ 0 & 10 & 5 & 0 & 0 & 10 \end{pmatrix}$	$\begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 12 \end{pmatrix}$
8	$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 2\pi & 4\pi^2 & 8\pi^3 & 16\pi^4 & 32\pi^5 & 64\pi^6 & 128\pi^7 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 4\pi & 12\pi^2 & 32\pi^3 & 80\pi^4 & 192\pi^5 & 448\pi^6 \\ 1 & \pi/2 & \pi^2/4 & \pi^3/8 & \pi^4/16 & \pi^5/32 & \pi^6/64 & \pi^7/128 \\ 0 & \pi/2 & \pi & 3\pi^2/4 & \pi^3/2 & 5\pi^4/16 & 3\pi^5/16 & 7\pi^6/64 \\ 1 & \pi & \pi^2 & \pi^3 & \pi^4 & \pi^5 & \pi^6 & \pi^7 \\ 0 & \pi & 2\pi & 3\pi^2 & 4\pi^3 & 5\pi^4 & 6\pi^5 & 7\pi^6 \end{pmatrix}$	$\begin{pmatrix} 4 \\ 4 \\ 0 \\ 0 \\ 2 \\ -2 \\ 1 \\ 0 \end{pmatrix}$

Remarque 4.16 : *Les tests 5 et 6 sont des variations du test 4 : $A_5 = A_4 + \Delta A$ et $b_6 = b_4 + \Delta b$. Ces 2 tests sont effectués pour évaluer les conséquences sur la solution x d'une perturbation ΔA de A_4 et d'une perturbation Δb de b_4.*

$$\Delta A = \begin{pmatrix} 0 & 0 & 0.1 & 0.2 \\ 0.08 & 0.04 & 0 & 0 \\ 0 & -0.02 & -0.11 & 0 \\ -0.01 & -0.01 & 0 & -0.02 \end{pmatrix}$$

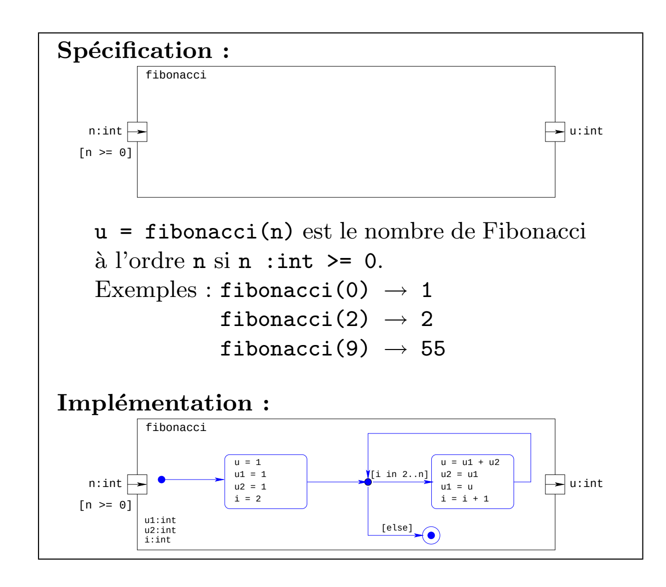
$$\Delta b = \begin{pmatrix} 0.01 \\ -0.01 \\ 0.01 \\ -0.01 \end{pmatrix}$$

Le test 7 correspond à l'exemple du pont de Wheatstone (figure 4.20). Le test 8 correspond à la détermination de la forme d'une came rotative qui doit mettre en mouvement un axe suivant une loi horaire donnée (exemple tiré de [8]).

Annexe A

Grands classiques

Cette annexe recense quelques algorithmes « historiques » dont les principales vertus aujourd'hui sont d'ordre pédagogique. A ce titre, tout informaticien débutant doit être capable de les redéfinir lui-même. Pour simplifier la lecture de ces algorithmes, ils sont présentés sous forme de fonctions PYTHON pour lesquelles les descriptions, les préconditions et les tests unitaires sont volontairement omis.



Algorithmes

Algorithme d'Euclide

Remarque A.1 : *Voir aussi les exemples 2.3 page 45, 2.9 page 51, 2.13 page 55, la figure 2.16 page 56 et le TD 2.15 page 56.*

L'algorithme d'Euclide concerne le calcul du plus grand commun diviseur (pgcd) de 2 entiers. Le pgcd de 2 entiers a et b peut se calculer en appliquant la relation de récurrence $\text{pgcd}(a, b) = \text{pgcd}(b, a\%b)$ jusqu'à ce que le reste $(a\%b)$ soit nul. Cette récurrence se traduit par l'algorithme récursif suivant :

```
def pgcd(a,b):
    if b == 0: d = a
    else: d = pgcd(b,a%b)
    return d
```

```
>>> pgcd(0,9)
9
>>> pgcd(12,18)
6
```

L'algorithme ci-dessous est la version itérative de cet algorithme récursif.

Algorithme A.1 – **Algorithme d'Euclide**

```
def pgcd(a,b):
    while b != 0:
        r = a % b
        a = b
        b = r
    return a
```

Coefficients du binôme

Remarque A.2 : *Voir aussi les TD 2.34 page 71 et 3.18 page 131.*

Remarque A.3 : *Le coefficient binomial* $\begin{pmatrix} n \\ k \end{pmatrix}$ *(ou* C_n^k, *combinaison de* k *parmi* n*) se lit « k parmi n ».*

Il s'agit ici de développer le binôme $(a + b)^n$ et de déterminer les coefficients des termes $a^{n-k}b^k$.

$$(a+b)^n = \sum_{k=0}^{n} \begin{pmatrix} n \\ k \end{pmatrix} a^{n-k}b^k = \sum_{k=0}^{n} \frac{n!}{k!(n-k)!} a^{n-k}b^k$$

La formule de Pascal lie les coefficients binomiaux entre eux selon la relation de récurrence :

$$\begin{pmatrix} n \\ k \end{pmatrix} = \begin{pmatrix} n-1 \\ k \end{pmatrix} + \begin{pmatrix} n-1 \\ k-1 \end{pmatrix} \text{ pour } (0 < k < n) \text{ et } \begin{pmatrix} n \\ 0 \end{pmatrix} = \begin{pmatrix} n \\ n \end{pmatrix} = 1$$

Elle permet un calcul rapide des coefficients pour les petites valeurs de n sans division ni multiplication. Elle est utilisée pour construire *à la main* le triangle de Pascal (remarque A.4) et c'est encore elle que est implémentée dans la version récursive du calcul des coefficients du binôme.

```
def coefBinome(n,k):
    if k == 0 or n == 0 or n == k:
        c = 1
    else:
        c = coefBinome(n-1,k) + coefBinome(n-1,k-1)
    return c
```

```
>>> coefBinome(2,1)
1
>>> coefBinome(7,4)
35
>>> coefBinome(14,7)
3432
```

La version itérative ci-dessous calcule directement le numérateur `num` et le dénominateur `den` du coefficient `c` en tenant compte de la simplification suivante :

$$\left(\begin{array}{c} n \\ k \end{array} \right) = \frac{n!}{k!(n-k)!} = \frac{n \cdot (n-1) \cdots (n-k+2) \cdot (n-k+1)}{1 \cdot 2 \cdot 3 \cdots k}$$

Remarque A.4 : *Triangle de Pascal*

n	C_n^k $(0 \leq k \leq n)$									
0	1									
1	1	1								
2	1	2	1							
3	1	3	3	1						
4	1	4	6	4	1					
5	1	5	10	10	5	1				
6	1	6	15	20	15	6	1			
7	1	7	21	35	35	21	7	1		
8	1	8	28	56	70	56	28	8	1	
9	1	9	36	84	126	126	84	36	9	1
⋮	⋮									

Algorithme A.2 – **Coefficients du binôme**

```
def coefBinome(n,k):
    num, den = 1, 1
    for i in range(1,k+1):
        num = num*(n-i+1)
        den = den*i
    c = num/den
    return c
```

Conversion en base b

Un entier n en base b est représenté par une suite de chiffres $(r_m r_{m-1} \ldots r_1 r_0)_b$ où les r_i sont des chiffres de la base b $(0 \leq r_i < b)$. Ce nombre n a pour valeur $\sum_{i=0}^{i=m} r_i b^i$.

Remarque A.5 : *Voir aussi l'exemple 3.1 page 100 et les TD 3.1 page 100, 3.3 page 112 et 3.4 page 113.*

$$n = r_m b^m + r_{m-1} b^{m-1} + \ldots + r_1 b^1 + r_0 b^0 = \sum_{i=0}^{i=m} r_i b^i$$

Nous choisirons ici de représenter un tel nombre par une liste $[r_m, r_{m-1}, \ldots, r_1, r_0]$. L'algorithme de conversion consiste à diviser successivement le nombre n par la base b $(n = bq + r)$ tant

que le quotient q n'est pas nul. L'ordre inverse des restes des différentes divisions (du dernier au premier reste, écrits de gauche à droite) donne la représentation du nombre n en base b.

$$\begin{aligned}
n &= q_0 b + r_0 = q_0 b^1 + r_0 b^0 \\
n &= (q_1 b^1 + r_1) b^1 + r_0 b^0 = q_1 b^2 + r_1 b^1 + r_0 b^0 \\
n &= (q_2 b^1 + r_2) b^2 + r_1 b^1 + r_0 b^0 = q_2 b^3 + r_2 b^2 + r_1 b^1 + r_0 b^0 \\
n &= \ldots \\
n &= 0 \cdot b^{m+1} + r_m b^m + \ldots + r_2 b^2 + r_1 b^1 + r_0 b^0 = \sum_{i=0}^{i=m} r_i b^i \\
n &= (r_m r_{m-1} \ldots r_1 r_0)_b
\end{aligned}$$

```
>>> conversion(23,10)
[2, 3]
>>> conversion(23,2)
[1, 1, 1, 0, 1]
>>> conversion(23,5)
[3, 4]
>>> conversion(23,12)
[1, 11]
>>> conversion(23,16)
[1, 7]
```

Algorithme A.3 – **Conversion en base b**

```
def conversion(n,b):
    code = []
    if n == 0:
        code = [0]
    while n != 0:
        code.insert(0,n%b)
        n = n/b
    return code
```

Courbe fractale de Koch

Remarque A.6 : *Voir aussi le TD 3.12 page 126.*

La courbe de von Koch est l'une des premières courbes fractales à avoir été décrite par le mathématicien suédois Helge von Koch (1870-1924). On peut la créer à partir d'un segment de droite, en modifiant récursivement chaque segment de droite de la façon suivante :

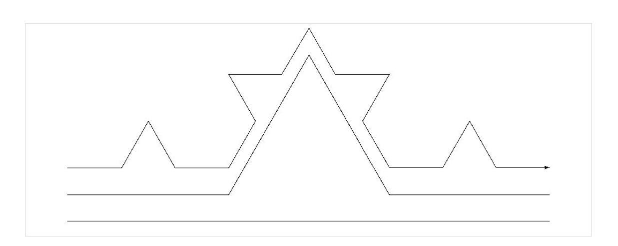
1. on divise le segment de droite en trois segments de longueurs égales,
2. on construit un triangle équilatéral ayant pour base le segment médian de la première étape,
3. on supprime le segment de droite qui était la base du triangle de la deuxième étape.

Avec les instructions *à la* Logo (voir annexe 2.6.1 page 91), on obtient l'algorithme récursif ci-dessous pour dessiner de telles courbes (figure A).

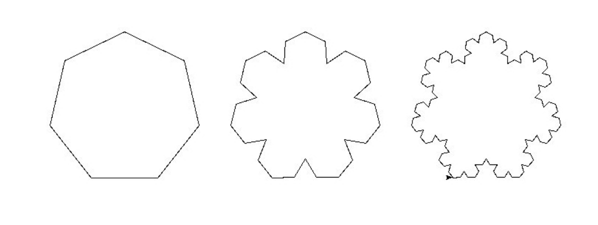
Algorithme A.4 – **Courbe fractale de Koch**

```
def kock(n,d):
    if n == 0: forward(d)
    else:
        kock(n-1,d/3.)
        left(60)
        kock(n-1,d/3.)
        right(120)
        kock(n-1,d/3.)
        left(60)
        kock(n-1,d/3.)
    return
```

Courbes de von Kock pour $n = 0$ (courbe du bas), $n = 1$ (courbe du milieu) et $n = 2$ (courbe du haut).



Les flocons de Koch s'obtiennent de la même façon que les courbes fractales, en partant d'un polygone régulier au lieu d'un segment de droite.
Ci-dessous, les flocons heptagonaux pour $n = 0$ (flocon de gauche), $n = 1$ (flocon du milieu) et $n = 2$ (flocon de droite).



Crible d'Eratosthène

Le crible d'Ératosthène (mathématicien grec du 3ème siècle avant JC) est un algorithme pour trouver tous les nombres premiers inférieurs à un certain entier naturel donné n. On forme une liste avec tous les entiers naturels compris entre 2 et n (`range(2,n+1)`) et on supprime les uns après les autres, les entiers qui ne sont pas premiers de la manière suivante : on commence par le début de la liste et dès que l'on trouve un entier qui n'a pas encore été supprimé, il est déclaré premier, on le supprime ainsi que tous les autres multiples de celui-ci et on recommence tant que la liste n'est pas vide. L'algorithme récursif suivant le généralise à toutes listes, triées par ordre croissant, d'entiers supérieurs ou égaux à 2.

```
def crible(t):
    if t[0]**2 > t[-1]: premiers = t
    else:
        premiers = []
        for element in t:
            if element%t[0] != 0:
                premiers.append(element)
        premiers = t[:1] + crible(premiers)
    return premiers
```

```
>>> crible(range(2,14))
[2, 3, 5, 7, 11, 13]
>>> crible(range(2,100))
[2, 3, 5, 7, 11, 13, 17, 19, 23,
 29, 31, 37, 41, 43, 47, 53, 59,
 61, 67, 71, 73, 79, 83, 89, 97]
>>> crible(range(2,14,3))
[2, 5, 11]
```

Un algorithme itératif correspondant est présenté ci-dessous.

Algorithme A.5 – **Crible d'Eratostène**

```
def crible(t):
    i = 0
    while i < len(t):
        j = i+1
        while j < len(t):
            if t[j]%t[0] == 0: del t[j]
            else: j = j + 1
        i = i + 1
    return t
```

Développement limité de la fonction $\sin(x)$

Remarque A.7 : *Voir aussi l'exemple 2.12 page 54 et les TD 2.14 page 55 et 2.40 page 73.*

Les développements limités des fonctions usuelles s'écrivent sous la forme de développements en série entière $\sum_{k=0}^{n} u_k$ comme par exemple pour la fonction sinus :

$$\sin(x) \approx \sum_{k=0}^{n} u_k = \sum_{k=0}^{n} (-1)^k \frac{x^{2k+1}}{(2k+1)!} = x - \frac{x^3}{6} + \frac{x^5}{120} + \ldots + (-1)^n \frac{x^{2n+1}}{(2n+1)!}$$

Ces développements font souvent intervenir les fonctions puissance et factorielle qui sont très « gourmandes » en temps de calcul ; on cherche alors à éviter leur utilisation en s'appuyant sur

une relation de récurrence entre les u_k. Ainsi, pour la fonction sinus :

$$u_{k+1} = (-1)^{k+1}\frac{x^{2(k+1)+1}}{(2(k+1)+1)!} = -(-1)^k\frac{x^{2k+1}}{(2k+1)!}\frac{x^2}{(2k+2)(2k+3)} = -u_k\frac{x^2}{(2k+2)(2k+3)}$$

Ce qui conduit à l'algorithme itératif ci-dessous.

Algorithme A.6 – **Développement limité**

```
def sinus(x,n):
    u = x
    y = u
    for k in range(0,n+1):
        u = -((x*x)/((2*k+2)*(2*k+3)))*u
        y = y + u
    return y
```

```
>>> sinus(pi/6,3)
0.50000000002027989
>>> sinus(pi/2,3)
1.0000035425842861
>>> sinus(pi,3)
0.0069252707075050518
```

```
>>> sinus(pi/6,7)
0.49999999999999994
>>> sinus(pi/2,7)
1.0000000000000437
>>> sinus(pi,7)
2.2419510632012503e-08
```

```
>>> sinus(pi/6,70)
0.49999999999999994
>>> sinus(pi/2,70)
1.0000000000000002
>>> sinus(pi,70)
2.4790606856821868e-16
```

Remarque A.8 : *Rappels :*

$$\sin(\frac{\pi}{6}) = \frac{1}{2} \qquad \sin(\frac{\pi}{2}) = 1 \qquad \sin(\pi) = 0$$

Fonction factorielle

La fonction factorielle qui calcule le produit des n premiers entiers positifs est simplement définie par la relation de récurrence :

$$\begin{cases} 0! = 1 \\ n! = n \cdot (n-1)! \quad \forall n \in N^* \end{cases} \qquad \left(n! = \prod_{k=1}^{n} k\right)$$

Remarque A.9 : *Voir aussi l'exemple 3.6 page 124 et les TD 2.13 page 54 et 2.23 page 62.*

Cette relation de récurrence est implémentée de manière récursive de la manière suivante :

```
def factorielle(n):
    if n < 2: u = 1
    else: u = n * factorielle(n-1)
    return u
```

```
>>> factorielle(0)
1
>>> factorielle(5)
120
>>> factorielle(10)
3628800
```

L'algorithme itératif du calcul de $n!$ est donné ci-dessous.

Algorithme A.7 – **Fonction factorielle**

```
def factorielle(n):
    u = 1
    for i in range(2,n+1):
        u = u * i
    return u
```

Fonction puissance

Remarque A.10 : *Voir aussi l'exemple 2.11 page 54 et le TD 3.17 page 131.*

La puissance entière p d'un nombre x est définie par : $p = x^n = \prod_{k=1}^{n} x = \underbrace{x \cdot x \cdot x \cdots x}_{n \text{ fois}}$. On peut la définir de manière récursive comme ci-dessous :

```
def puissance(x,n):
    if n == 0: p = 1
    else: p = x * puissance(x,n-1)
    return p
```

```
>>> puissance(2,0)
1
>>> puissance(2,20)
1048576
```

L'algorithme itératif du calcul de x^n est le suivant.

Algorithme A.8 – **Fonction puissance**

```
def puissance(x,n):
    p = x
    for k in range(1,n):
        p = p*x
    return p
```

Nombres de Fibonacci

Les nombres de Fibonacci sont donnés par la suite définie par la relation de récurrence ci-dessous :

$$\begin{cases} f_0 = f_1 = 1 \\ f_n = f_{n-1} + f_{n-2} \ \forall n > 1 \end{cases}$$

Les 10 premiers nombres de la suite de Fibonacci valent donc successivement $f_0 = 1$, $f_1 = 1$, $f_2 = 2$, $f_3 = 3$, $f_4 = 5$, $f_5 = 8$, $f_6 = 13$, $f_7 = 21$, $f_8 = 34$, et $f_9 = 55$.

Le $n^{ème}$ nombre de Fibonacci peut donc se calculer de manière récursive en appliquant simplement la définition de la suite de Fibonacci.

```
def fibonacci(n):
    if n < 2: f = 1
    else: f = fibonacci(n-1) + fibonacci(n-2)
    return f
```

```
>>> for n in range(0,10):
        print fibonacci(n),

1 1 2 3 5 8 13 21 34 55
```

Un algorithme itératif du calcul du $n^{ème}$ nombre de Fibonacci est présenté ci-dessous.

Remarque A.11 : *Voir aussi l'exemple 3.4 page 104, les figures 3.13 page 112 et 3.14 page 122 et la remarque 3.3 page 113.*

Algorithme A.9 – **Nombres de Fibonacci**

```
def fibonacci(n):
    f,f1,f2 = 2,1,1
    for i in range(3,n+1) :
        f2 = f1
        f1 = f
        f = f1 + f2
    return f
```

Palindrome

Un palindrome est une séquence dont l'ordre des éléments reste le même qu'on la parcourt du premier au dernier ou du dernier au premier. Ainsi, `[1,2,2,1]`, `"kayak"` sont des palindromes au même titre que l'expression arithmétique `1234+8765=9999=5678+4321`. Dans les chaînes de caractères, si l'on ne prend pas en compte la casse (minuscules ou majuscules), les espaces, les signes de ponctuation et les signes diacritiques (accents, cédilles), alors `"engage le jeu que je le gagne"` et `"A man, a plan, a canal : Panama"` sont également des palindromes.

Algorithme A.10 – **Palindrome**

```
def palindrome(t):
    n, i, ok = len(t), 0, True
    while i < n/2 and ok == True:
        if t[i] != t[n-1-i]: ok = False
        else: i = i + 1
    return ok
```

Produit de matrices

Remarque A.12 : *Voir aussi le TD 4.12 page 170.*

Remarque A.13 : *Le produit de deux matrices n'est défini que si le nombre r de colonnes de la première matrice est égal au nombre de lignes de la deuxième matrice.*

Le produit C de 2 matrices A et B respectivement de dimensions (n, r) et (r, m) est tel que :

$$c_{i,j} = \sum_{k=0}^{r-1} a_{ik} \cdot b_{kj}$$

Algorithme A.11 – **Produit de matrices**

```
def produitMatrice(a,b):
    c = []
    n,r,m = len(a),len(a[0]),len(b[0])
    for i in range(n):
        c.append([])
        for j in range(m):
            x = 0
            for k in range(r):
                x = x + a[i][k]*b[k][j]
            c[i].append(x)
    return c
```

```
>>> a = [[1,2],[3,4]]
>>> b = [[2,1],[-1,-2]]
>>> produitMatrice(a,b)
[[0, -3], [2, -5]]
```

```
>>> a = [[1,2,3,4]]
>>> b = [[1],[2],[3],[4]]
>>> produitMatrice(a,b)
[[30]]
```

Recherche dichotomique

Il s'agit d'un algorithme de recherche d'un élément `x` dans une liste `t` déjà triée. Le principe de la recherche dichotomique consiste à comparer `x` avec l'élément `t[m]` du milieu de la liste `t` triée :

- si `x == t[m]`, on a trouvé une solution et la recherche s'arrête ;
- si `x < t[m]`, `x` ne peut se trouver que dans la moitié gauche de la liste `t` puisque celle-ci est triée par ordre croissant ; on poursuit alors la recherche de la même manière uniquement dans la moitié gauche de la liste ;
- si `x > t[m]`, `x` ne peut se trouver que dans la moitié droite de la liste `t` ; on poursuit alors la recherche uniquement dans la moitié droite de la liste.

Remarque A.14 : *Voir aussi la section 4.3.2 page 172 consacrée à la recherche dichotomique.*

Algorithme A.12 – **Recherche dichotomique**

```
def rechercheDichotomique(t,x):
    ok, m = False, (gauche + droite)/2
    while gauche <= droite and not ok:
        m = (gauche + droite)/2
        if t[m] == x: ok = True
        elif t[m] > x: droite = m - 1
        else: gauche = m + 1
    return ok,m
```

Recherche séquentielle

La recherche séquentielle d'un élément `x` dans une liste `t` consiste à comparer l'élément recherché `x` successivement à tous les éléments de la liste `t` jusqu'à trouver une correspondance.

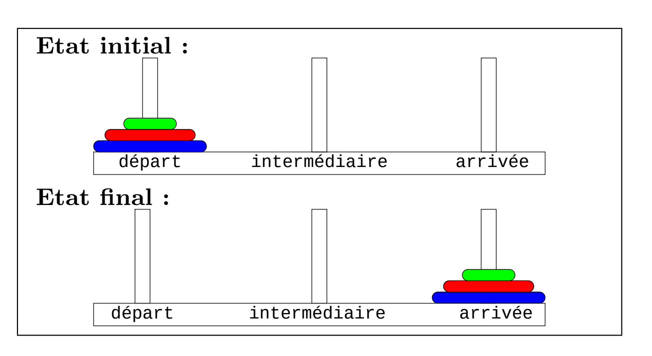
Remarque A.15 : *Voir aussi la section 4.3.1 page 171 consacrée à la recherche séquentielle.*

Algorithme A.13 – **Recherche séquentielle**

```
def rechercheSequentielle(t,x,debut,fin):
    assert type(t) is list
    assert 0 <= debut <= fin < len(t)
    ok, r = False, debut
    while r <= fin and not ok:
```

```
        if t[r] == x: ok = True
        else: r = r + 1
    return ok,r
```

Remarque A.16 : *Voir aussi l'exemple 3.5 page 123 et le TD 3.9 page 123.*



Tours de Hanoï

Les « tours de Hanoï » est un jeu qui consiste à déplacer n disques de diamètres différents d'une tour de « départ » à une tour d'« arrivée » en passant par une tour « intermédiaire » et ceci en un minimum de coups, tout en respectant les règles suivantes :

- on ne peut déplacer qu'un disque à la fois,
- on ne peut placer un disque que sur un autre disque plus grand que lui ou sur une tour vide.

Dans l'état initial, les n disques sont placés sur la tour « départ ». Dans l'état final, tous les disques se retrouvent placés dans le même ordre sur la tour « arrivée ».

Algorithme A.14 – **Tours de Hanoï**

```
def hanoi(n,gauche,milieu,droite):
    if n > 0:
        hanoi(n-1,gauche,droite,milieu)
        deplacer(n,gauche,droite)
        hanoi(n-1,milieu,droite,gauche)
    return
```

Si le déplacement se traduit par un simple affichage du type : déplacer disque « n » de la tour « gauche » à la tour « droite », alors l'appel de la procédure `hanoi` pour $n = 3$ produit les sorties suivantes.

```
def deplacer(n,gauche,droite):
    print 'déplacer disque', n,
          'de la tour', gauche,
          'à la tour', droite
    return
```

```
>>> hanoi(3,'d','i','a')
déplacer disque 1 de la tour d à la tour a
déplacer disque 2 de la tour d à la tour i
déplacer disque 1 de la tour a à la tour i
déplacer disque 3 de la tour d à la tour a
déplacer disque 1 de la tour i à la tour d
déplacer disque 2 de la tour i à la tour a
déplacer disque 1 de la tour d à la tour a
```

Tri bulles

Dans le tri bulles, on parcourt la liste en commençant par la fin, en effectuant un échange à chaque fois que l'on trouve deux éléments successifs qui ne sont pas dans le bon ordre.

Remarque A.17 : *Voir aussi le TD 4.28 page 184.*

Remarque A.18 : *Le nom du tri bulles vient de ce que les éléments les plus petits (les plus « légers ») remontent vers le début de la liste comme des bulles vers le haut d'une bouteille d'eau gazeuse.*

Algorithme A.15 – **Tri bulles**

```
def triBulles(t,debut,fin):
    while debut < fin:
        for j in range(fin,debut,-1):
            if t[j] < t[j-1]:
                t[j],t[j-1] = t[j-1],t[j]
        debut = debut + 1
    return t
```

```
>>> s = [9,8,7,6,5,4]
>>> triBulles(s,0,len(s)-1)
[4, 5, 6, 7, 8, 9]
```

```
>>> s = [9,8,7,6,5,4]
>>> triBulles(s,1,4)
[9, 5, 6, 7, 8, 4]
```

Tri fusion

Dans le tri fusion, on partage la liste à trier en deux sous-listes que l'on trie, et on interclasse (on fusionne) ces deux sous-listes.

Algorithme A.16 – **Tri fusion**

```
def triFusion(t,debut,fin):
    if debut < fin:
        m = (debut + fin)/2
        triFusion(t,debut,m)
        triFusion(t,m+1,fin)
        fusion(t,debut,m,fin)
    return t
```

La fusion consiste à construire, à partir des deux sous-listes triées (la première de l'indice `debut` à l'indice `k`, la deuxième de l'indice `k+1` à l'indice `fin`) une liste elle-même triée des indices `debut` à `fin` et contenant l'ensemble des éléments des deux sous-listes d'origine.

Fusion

```
def fusion(t,debut,k,fin):
    i1, i2 = debut, k+1
    while i1 <= k and i2 <= fin:
        if t[i2] >= t[i1]: i1 = i1 + 1
        else:
            t[i1], t[i1+1:i2+1] = t[i2], t[i1:i2]
            k = k + 1
            i2 = i2 + 1
    return t
```

```
>>> s = [4,5,6,1,2,3]
>>> fusion(s,0,2,len(s)-1)
[1, 2, 3, 4, 5, 6]
```

```
>>> s = [4,5,6,1,2,3]
>>> fusion(s,1,2,4)
[4, 1, 2, 5, 6, 3]
```

Tri par insertion

Remarque A.19 : *Voir aussi la section 4.4.2 page 175 consacrée au tri par insertion.*

Dans le tri par insertion, on trie successivement les premiers éléments de la liste : à la $i^{ème}$ étape, on insère le $i^{ème}$ élément à son rang parmi les $i-1$ éléments précédents qui sont déjà triés entre eux.

Algorithme A.17 – **Tri par insertion**

```
def triInsertion(t,debut,fin):
    assert type(t) is list
    assert 0 <= debut <= fin < len(t)
    for k in range(debut+1,fin+1):
        i = k - 1
        x = t[k]
        while i >= debut and t[i] > x:
            t[i+1] = t[i]
            i = i - 1
        t[i+1] = x
    return t
```

Tri par sélection

Le tri par sélection d'une liste consiste à rechercher le minimum de la liste à trier, de le mettre en début de liste en l'échangeant avec le premier élément et de recommencer sur le reste de la liste.

Remarque A.20 : *Voir aussi la section 4.4.1 page 174 consacrée au tri par sélection.*

Algorithme A.18 – **Tri par sélection**

```
def triSelection(t,debut,fin):
    assert type(t) is list
    assert 0 <= debut <= fin < len(t)
    while debut < fin:
        mini = minimum(t,debut,fin)
        t[debut],t[mini] = t[mini],t[debut]
        debut = debut + 1
    return t
```

Tri rapide

Le principe du tri rapide est le suivant : on partage la liste à trier en deux sous-listes telles que tous les éléments de la première soient inférieurs à tous les éléments de la seconde. Pour partager la liste en deux sous-listes, on choisit un des éléments de la liste (par exemple le premier) comme pivot. On construit alors une sous-liste avec tous les éléments inférieurs ou égaux à ce pivot et une sous-liste avec tous les éléments supérieurs au pivot. On trie les deux sous-listes selon le même processus jusqu'à avoir des sous-listes réduites à un seul élément.

Remarque A.21 : *Voir aussi la section 4.4.3 page 177 consacrée au tri rapide.*

Algorithme A.19 – **Tri rapide**

```
def triRapide(t,debut,fin):
    assert type(t) is list
    assert 0 <= debut
    assert fin <= len(t)
    if debut < fin:
        pivot = t[debut]
        place = partition(t,debut,fin,pivot)
```

```
        triRapide(t,debut,place-1)
        triRapide(t,place+1,fin)
    return t
```

Partition

```
def partition(t,debut,fin,pivot):
    place,inf,sup = debut,debut,fin;
    while place <= sup:
        if t[place] == pivot: place = place + 1
        elif t[place] < pivot:
            t[inf],t[place] = t[place],t[inf]
            inf = inf + 1
            place = place + 1
        else:
            t[sup],t[place] = t[place],t[sup]
            sup = sup - 1
    if place > 0: place = place - 1
    return place
```

```
>>> s = [3,6,5,4,1,2]
>>> partition(s,0,len(s)-1,4), s
(3, [3, 2, 1, 4, 5, 6])
```

```
>>> s = [3,6,5,4,1,2]
>>> partition(s,1,4,4), s
(2, [3, 1, 4, 5, 6, 2])
```

Annexe B

Travaux dirigés

Le planning prévisionnel de la section 1.6.3 page 34 permet de visualiser la répartition des 7 séances de travaux dirigées organisées au cours des 15 semaines du cours d'Informatique S1 de l'ENIB. Dans cette annexe, on précise pour chaque TD

1. les objectifs recherchés,
2. les exercices de TD à préparer avant la séance,
3. les exercices complémentaires pour s'exercer.

TD 1

Objectifs

1. Prise en main de l'environnement informatique (système d'exploitation Linux, environnement de programmation Python, site Web) 1h30

2. Exploitation des instructions de base : affectation et tests (chapitre 2, sections 2.2 et 2.3 page 46) 1h30

Exercices de TD

Exercices complémentaires

TD 2

Objectifs

1. Exploitation des instructions de base : boucles (chapitre 2, section 2.4) 3h

Exercices de TD

Exercices complémentaires

TD 2.41 Tables de vérité. 74

Analyser

TD 2.44 Zéro d'une fonction. 75

TD 3

Objectifs

1. Exploitation des instructions de base : affectation, tests et boucles imbriqués (chapitre 2 en entier) 3h

Exercices de TD

Il s'agit d'un TD récapitulatif sur les instructions de base, aussi les exercices à préparer font partie des exercices complémentaires des TD précédents.

TD 2.40 Développements limités. 73
TD 2.41 Tables de vérité. 74
TD 2.44 Zéro d'une fonction. 75

TD 4

Objectifs

1. Spécifier et implémenter des fonctions (chapitre 3, section 3.2) 3h

Exercices de TD

TD 3.1 Codage des entiers positifs (1). 100
TD 3.2 Codage d'un nombre fractionnaire. 101
TD 3.3 Décodage base b → décimal. 112
TD 3.4 Codage des entiers positifs (2). 113
TD 3.5 Une spécification, plusieurs implémentations. 114

Exercices complémentaires

Analyser

TD 5

Objectifs

1. Spécifier et implémenter des fonctions (chapitre 3, section 3.2) 3h

Exercices de TD

Il s'agit d'un TD récapitulatif sur la définition des fonctions, aussi les exercices à préparer font partie des exercices complémentaires des TD précédents.

TD 6

Objectifs

1. Appeler des fonctions itératives ou récursives (chapitre 3, section 3.3) 3h

Exercices de TD

Exercices complémentaires

TD 7

Objectifs

1. Manipulation de séquences (chapitre 4, section 4.2) 3h

Exercices de TD

Exercices complémentaires

Annexe C

Contrôles types

Cette annexe propose des exemples corrigés de contrôles d'autoformation (CAF) et de contrôles de compétences (DS) tels qu'ils sont programmés au cours du semestre S1 (voir le planning prévisionnel en section 1.6.3 page 34). L'esprit de ces contrôles est résumé dans le chapitre 1 d'introduction à la section 1.3.3 page 14.

Quel que soit le type de contrôle, un exercice cherche à évaluer un objectif particulier. Aussi, la notation exprimera la distance qui reste à parcourir pour atteindre cet objectif (figure C) :

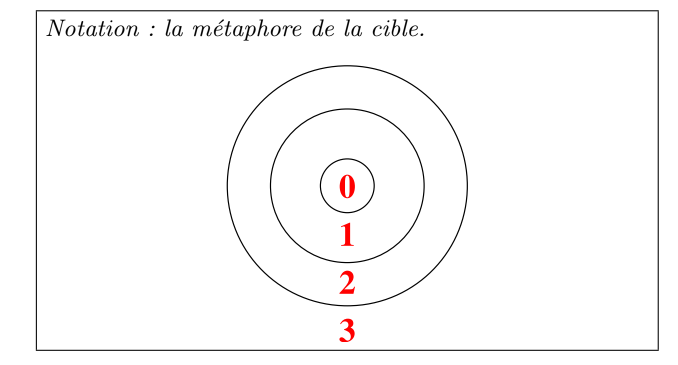
0 : « en plein dans le mille ! »	→ l'objectif est atteint
1 : « pas mal ! »	→ on se rapproche de l'objectif
2 : « juste au bord de la cible ! »	→ on est encore loin de l'objectif
3 : « la cible n'est pas touchée ! »	→ l'objectif n'est pas atteint

Ainsi, et pour changer de point de vue sur la notation, le contrôle est réussi lorsqu'on a 0 ! Il n'y a pas non plus de 1/2 point ou de 1/4 de point : le seul barème possible ne comporte que 4 niveaux : 0, 1, 2 et 3. On ne cherche donc pas à « grappiller » des points :

- on peut avoir 0 (objectif atteint) et avoir fait une ou deux erreurs bénignes en regard de l'objectif recherché ;
- on peut avoir 3 (objectif non atteint) et avoir quelques éléments de réponse corrects mais sans grand rapport avec l'objectif.

Sommaire

Notation : la métaphore de la cible.



Remarque C.1 : *Une absence à un contrôle conduit à la note 4 (« la cible n'est pas visée »).*

CAF1 : calculs booléens

Développement d'une expression booléenne

Développer l'expression booléenne suivante :

$$t = \overline{a \cdot b \cdot c \cdot d}$$

$$\begin{array}{rcll} t & = & \overline{a \cdot b \cdot c \cdot d} & \\ & = & \overline{(a \cdot b) \cdot (c \cdot d)} & \text{associativité} \\ & = & \overline{a \cdot b} + \overline{c \cdot d} & \text{De Morgan} \\ & = & (\overline{a} + \overline{b}) + (\overline{c} + \overline{d}) & \text{De Morgan} \\ & = & \overline{a} + \overline{b} + \overline{c} + \overline{d} & \text{associativité} \end{array}$$

Table de vérité d'une expression booléenne

Etablir la table de vérité de l'expression booléenne suivante :

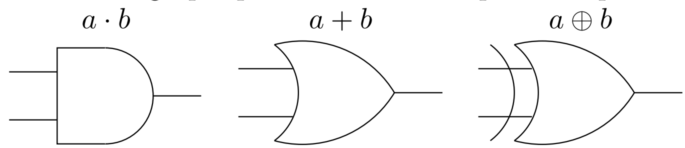
$$t = ((a \Rightarrow b) \cdot (b \Rightarrow c)) \Rightarrow (\overline{c} \Rightarrow \overline{a})$$

On pose $u = (a \Rightarrow b) \cdot (b \Rightarrow c)$:

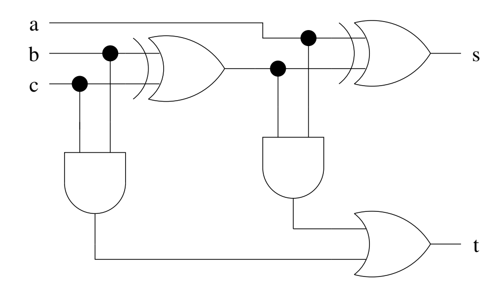
a	b	c	$(a \Rightarrow b)$	$(b \Rightarrow c)$	u	$\overline{c}$	$\overline{a}$	$(\overline{c} \Rightarrow \overline{a})$	t
0	0	0	1	1	1	1	1	1	1
0	0	1	1	1	1	0	1	1	1
0	1	0	1	0	0	1	1	1	1
0	1	1	1	1	1	0	1	1	1
1	0	0	0	1	0	1	0	0	1
1	0	1	0	1	0	0	0	1	1
1	1	0	1	0	0	1	0	0	1
1	1	1	1	1	1	0	0	1	1

Table de vérité d'un circuit logique

On considère les conventions graphiques traditionnelles pour les opérateurs logiques $\cdot$, $+$ et $\oplus$:



Etablir la table de vérité du circuit logique ci-dessous où a, b et c sont les entrées, x et y les sorties.



a	b	c	$(b \cdot c)$	$(b \oplus c)$	$(a \cdot (b \oplus c))$	x	y
0	0	0	0	0	0	0	0
0	0	1	0	1	0	1	0
0	1	0	0	1	0	1	0
0	1	1	1	0	0	0	1
1	0	0	0	0	0	1	0
1	0	1	0	1	1	0	1
1	1	0	0	1	1	0	1
1	1	1	1	0	0	1	1

Il s'agit de l'addition des 3 bits a, b et c : x est la somme et y la retenue

a	0	0	0	0	1	1	1	1
b	0	0	1	1	0	0	1	1
c	0	1	0	1	0	1	0	1
yx	00	01	01	10	01	10	10	11

CAF2 : codage des nombres

Représentation en complément à 2

1. Déterminer la plage de valeurs entières possibles lorsqu'un entier positif, négatif ou nul est codé en binaire sur $k = 4$ chiffres dans la représentation en complément à 2.

> Lors d'un codage en binaire sur k bits, seuls les nombres entiers relatifs n tels que $-2^{k-1} \leq n < 2^{k-1}$ peuvent être représentés ($n \in [-2^{k-1}; 2^{k-1}[$).
>
> Application numérique : $k = 4$: $[-2^{4-1}; 2^{4-1}[= [-8; +8[$

2. Coder l'entier $n = (-93)_{10}$ sur $k = 8$ chiffres en base $b = 2$ en utilisant la représentation du complément à 2.

> (a) coder $|n| = (93)_{10}$ sur $(k-1) = 7$ chiffres : $(93)_{10} = (1011101)_2$
> (b) mettre le bit de poids fort ($k^{ème}$ bit) à 0 : $(01011101)_2$
> (c) inverser tous les bits obtenus : $(01011101)_2 \rightarrow (10100010)_2$
> (d) ajouter 1 : $(10100010)_2 + (00000001)_2 = (10100011)_2$
> (e) conclusion : $n = (-93)_{10} = (10100011)_2$

3. Vérifier la réponse précédente en additionnant en base $b = 2$, n et $(-n)$, codés sur $k = 8$ chiffres dans la représentation du complément à 2.

$$\begin{array}{rcccc} & n & = & (-93)_{10} & = & (10100011)_2 \\ + & (-n) & = & (+93)_{10} & = & (01011101)_2 \\ \hline = & 0 & = & (0)_{10} & = & 1(00000000)_2 \end{array}$$

> Le bit de poids fort (le 9ème bit à partir de la droite : 1) est perdu.

IEEE 754 simple précision

Coder le nombre réel $x = -73.25$ selon la norme IEEE 754 en simple précision.

partie entière : : $(73)_{10} = (1001001)_2$

partie fractionnaire : $(0.25)_{10} = (0.01)_2$

mantisse normalisée : $(1001001.01)_2 = (1.00100101)_2 \cdot 2^6 = (1.00100101)_2 \cdot 2^{(00000110)_2}$

exposant relatif à 127 : $(01111111)_2 + (00000110)_2 = (10000101)_2$

signe : $(1)_2$

norme IEEE 754 :

1	1 0 0 0 0 1 0 1	0 0 1 0 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

CAF3 : recherche d'un élément

Recherche d'une occurrence

Définir une fonction `rechercheKieme` qui recherche le rang `r` de la `k`^{ème} occurrence d'un élément `x` à partir du début d'une liste `t`.
Exemple : `t = [7,4,3,2,4,4]`, `k = 2`, `x = 4` $\rightarrow$ `r = 4`

```python
def rechercheKieme(t,x,k):
    """
    recherche la kième occurrence de x dans le tableau t
    -> (found, index)
    (found == False and index == len(t)) or
    (found == True and 0 <= index < len(t))

    >>> rechercheKieme([],7,2)
    (False, 0)
    >>> rechercheKieme([1,2,3,2],7,2)
    (False, 4)
    >>> rechercheKieme([1,2,3,2],2,2)
    (True, 3)
    """
    assert type(t) is list
    assert type(k) is int
    found, index, occur = False, 0, 0
    while index < len(t) and not found:
        if t[index] == x:
            occur = occur + 1
            if occur == k: found = True
            else: index = index + 1
        else: index = index + 1
    return found, index
```

Exécution d'une fonction

On considère la fonction g ci-dessous.

```python
def g(t,x,k,d):
    assert type(t) is list
    assert type(k) is int and k > 0
    assert d in [0,1]

    ok, i, n = False, (1-d)*(len(t)-1), 0
    print i,n,ok #------------------------- affichage
    while i in range(0,len(t)) and not ok:
        if t[i] == x:
            n = n + 1
            if n == k: ok = True
            else: i = i - (-1)**d
        else: i = i - (-1)**d
        print i,n,ok #--------------------- affichage
    print i,n,ok #------------------------- affichage

    return ok,i
```

1. Qu'affiche l'appel `g(t,2,3,0)` quand `t = [5,2,5,3,5,2,5,2]` ?

```
>>> t = [5,2,5,3,5,2,5,2]
>>> g(t,2,3,0)
7 0 False
6 1 False
5 1 False
4 2 False
3 2 False
2 2 False
1 2 False
1 3 True
1 3 True
(True, 1)
```

2. Compléter la spécification de la fonction g.

Description :

```
"""
(ok,i) = g(t,x,k,d)
i est le rang de la kème occurence de x en partant
du début (d = 1) ou de la fin (d = 0) de la liste t,
si cette occurence existe (ok = True) ;
i = d*len(t) - (1-d) sinon (ok = False)
"""
```

Jeu de tests :

```
"""
>>> g([],7,2,0)
(False, -1)
>>> g([],7,2,1)
(False, 0)
>>> g([1,2,3,2,5],2,2,0)
(True, 1)
>>> g([1,2,3,2,5],2,2,1)
(True, 3)
>>> g([1,2,3,2,5],2,3,0)
(False, -1)
>>> g([1,2,3,2,5],2,3,1)
(False, 5)
"""
```

DS1 : instructions de base

Exécution d'une séquence d'instructions

Qu'affiche la séquence d'instructions suivante ?

```
a = 289
x = 1
z = a
y = 0
t = x
print a,x,z,t,y
while x <= a: x = x*4

print a,x,z,t,y
t = x
while x > 1:
  x = x/4
  t = t/2 - x
  if t <= z:
    z = z - t
    t = t + x*2
  y = t/2
  print a,x,z,t,y

print a,x,z,t,y
```

a	x	z	t	y
289	1	289	1	0
289	1024	289	1	0
289	256	33	768	384
289	64	33	320	160
289	16	33	144	72
289	4	33	68	34
289	1	0	35	17
289	1	0	35	17

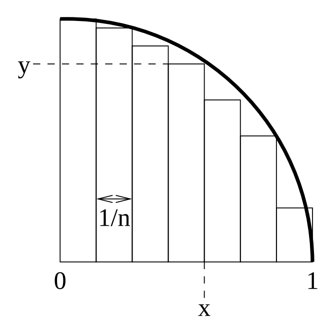
Il s'agit du calcul de la racine carrée entière y d'un nombre entier a : $y = 17 = \sqrt{289} = \sqrt{a}$.

Calcul de π

Dans cette section, on se propose de calculer π selon la méthode des rectangles.

Selon cette méthode, on calcule π à partir de l'expression de la surface S d'un cercle de rayon unité. On approche la surface du quart de cercle par n rectangles d'aire $A_i = y_i/n$.

Ecrire un algorithme qui calcule π selon la méthode des rectangles à l'ordre n.



Les points du cercle de rayon unité ont pour coordonnées $x_k = \dfrac{k}{n}$ et $y_k = \sqrt{1 - \dfrac{k^2}{n^2}}$ et un rectangle de base a pour surface $s_k = \dfrac{1}{n}\sqrt{1 - \dfrac{k^2}{n^2}}$. La surface du 1/4 de cercle s'écrit alors comme la somme des rectangles de base : $S = \dfrac{\pi}{4} = \displaystyle\sum_{k=0}^{n} s_k = \dfrac{1}{n}\sum_{k=0}^{n}\sqrt{1 - \dfrac{k^2}{n^2}}$.

Ce qui donne avec l'interpréteur `python` :

```
>>> from math import *
>>> n = 20000000
>>> y = 0.
>>> for k in range(0,n+1) :
...    y = y + sqrt(1 - (1.*k*k)/(n*n))
...
>>> pi - 4*y/n
-9.9986801060936159e-08
```

Zéro d'une fonction

Remarque C.2 : *Voir TD 2.44 page 75.*

Dans cette section, on recherche le zéro d'une fonction f continue sur un intervalle $[a, b]$ telle que $f(a).f(b) < 0$ (il existe donc une racine de f dans $]a, b[$ que nous supposerons unique).

Ecrire un algorithme qui détermine le zéro de $\cos(x)$ dans $[1, 2]$ selon la méthode des tangentes.

Indications : soit x_n une approximation de la racine c recherchée : $f(c) = f(x_n) + (c - x_n)f'(x_n)$; comme $f(c) = 0$, on a : $c = x_n - f(x_n)/f'(x_n)$. Posons $x_{n+1} = x_n - f(x_n)/f'(x_n)$: on peut considérer que x_{n+1} est une meilleure approximation de c que x_n. On recommence le procédé avec x_{n+1} et ainsi de suite jusqu'à ce que $|x_{n+1} - x_n|$ soit inférieur à un certain seuil s.

```
>>> from math import *
>>> x1 = 1.
>>> x2 = 2.
>>> s = 1.e-9
>>> f = cos
>>> df = sin
>>> x = x2 - f(x2)/(-df(x2))
>>> while fabs(x-x2) > s :
... x2 = x
... x = x - f(x)/(-df(x))
...
>>> cos(x)
6.1230317691118863e-17
```

Tableau d'Ibn al-Banna

L'exercice suivant est inspiré du premier chapitre du livre « Histoire d'algorithmes » [2]. On considère ici le texte d'Ibn al-Banna concernant la multiplication à l'aide de tableaux.

> Tu construis un quadrilatère que tu subdivises verticalement et horizontalement en autant de bandes qu'il y a de positions dans les deux nombres multipliés. Tu divises diagonalement les carrés obtenus, à l'aide de diagonale allant du coin inférieur gauche au coin supérieur droit.
>
> Tu places le multiplicande au-dessus du quadrilatère, en faisant correspondre chacune de ses positions à une colonne. Puis, tu places le multiplicateur à gauche ou à droite du quadrilatère, de telle sorte qu'il descende avec lui en faisant correspondre également chacune de ses positions à une ligne. Puis, tu multiplies, l'une après l'autre, chacune des positions du multiplicande du carré par toutes les positions du multiplicateur, et tu poses le résultat partiel correspondant à chaque position dans le carré où se coupent respectivement leur colonne et leur ligne, en plaçant les unités au-dessus de la diagonale et les dizaines en dessous. Puis, tu commences à additionner, en partant du coin supérieur gauche : tu additionnes ce qui est entre les diagonales, sans effacer, en plaçant chaque nombre dans

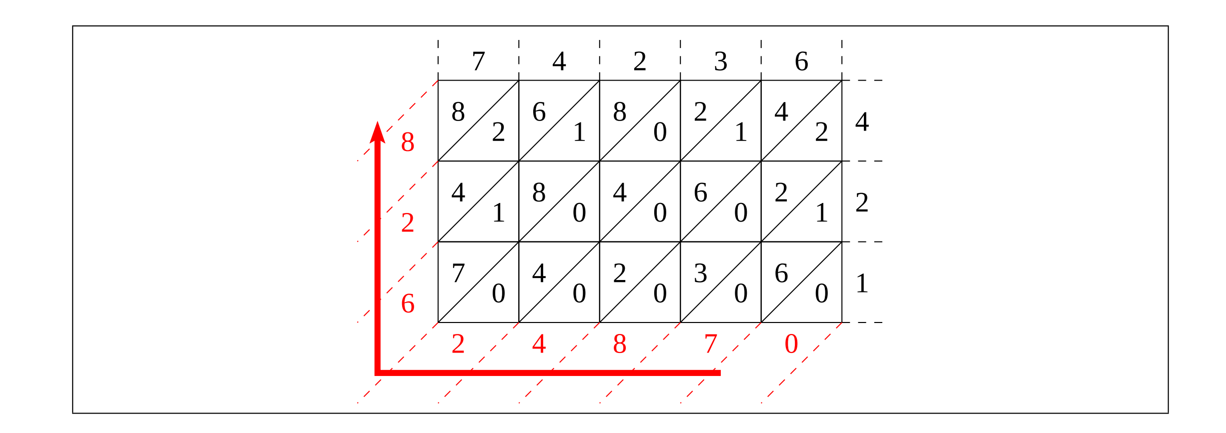
Remarque C.3 : *Voir TD 1.26 page 23.*

Remarque C.4 :

- *L'écriture du multiplicande s'effectue de droite à gauche (exemple : 352 s'écrira donc 253).*
- *L'écriture du multiplicateur s'effectue de bas en haut (exemple : $\begin{smallmatrix}3\\5\\2\end{smallmatrix}$ s'écrira donc $\begin{smallmatrix}2\\5\\3\end{smallmatrix}$).*

sa position, en transférant les dizaines de chaque somme partielle à la diagonale suivante et en les ajoutant à ce qui y figure.
La somme que tu obtiendras sera le résultat.

En utilisant la méthode du tableau d'Ibn al-Banna, calculer 63247×124 $(= 7842628)$.



DS2 : procédures et fonctions

Calcul de π

Définir une fonction qui calcule π à l'ordre n selon la formule :

$$\pi = 2 \cdot \frac{4}{3} \cdot \frac{16}{15} \cdot \frac{36}{35} \cdot \frac{64}{63} \cdots = 2 \prod_{k=1}^{n} \frac{4k^2}{4k^2 - 1}$$

```python
def pi(n):
    """
    y = pi(n)
    calcule pi à l'ordre n
    >>> abs(pi(1) - math.pi) < 1.
    True
    >>> abs(pi(100) - math.pi) < 1./100
    True
    >>> abs(pi(10000000) - math.pi) < 1.e-7
    True
    """
    assert type(n) is int and n > 0
    y = 2.
    for k in range(1,n+1):
        u = 4*k*k
        y = y*u/(u-1)
    return y
```

Conversion décimal $\rightarrow$ base b

Remarque C.5 : *Voir TD 3.4 page 113.*

Définir une fonction qui calcule le code t en base b sur k chiffres d'un entier n.

Exemples pour $n = 23$:

```
b = 2   k = 7 → t = [0, 0, 1, 0, 1, 1, 1]
b = 5   k = 5 → t = [0, 0, 0, 4, 3]
b = 21  k = 2 → t = [1, 2]
b = 25  k = 6 → t = [0, 0, 0, 0, 0, 23]
```

```python
def code(n,b,k):
    """
    c = code(n,b,k)
    code n en base b sur k chiffres
    >>> code(23,2,8)
    [0, 0, 0, 1, 0, 1, 1, 1]
    >>> code(23,2,7)
    [0, 0, 1, 0, 1, 1, 1]
    >>> code(23,5,5)
    [0, 0, 0, 4, 3]
    >>> code(23,21,2)
    [1, 2]
    >>> code(23,25,6)
    [0, 0, 0, 0, 0, 23]

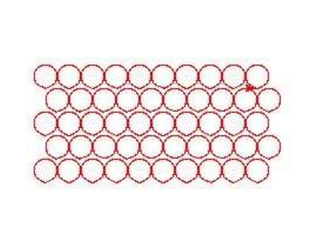
    """
    assert type(n) is int and n >= 0
    assert type(b) is int and b > 1
    assert type(k) is int and k > 0
    c = []
    for i in range(k): c.append(0)
    q = n
    i = k-1
    while q > 0 and i >= 0:
        r = q%b
        q = q/b
        c[i] = r
        i = i - 1
    return c
```

Quinconce

Définir une procédure qui dessine $n \times m$ cercles de rayon r disposés en quinconce sur n rangées de m cercles chacune. On utilisera les instructions de tracé *à la Logo*.

Remarque C.6 : *Voir TD 2.22 page 61.*

```
def quinconce(n,m,r):
    """
    quinconce(n,m,r)
    trace n rangées de m cercles de rayon r
    disposées en quinconce
    >>> quinconce(5,10,10)
    """
    assert type(n) is int and n > 0
    assert type(m) is int and m > 0
    assert type(r) is int and r > 0
    for i in range(n) :
        x0 = r*(i%2)
        y0 = 2*i*r
        for j in range(m) :
            up()
            goto(x0+2*j*r,y0)
            down()
            circle(r)
    return
```



Coefficients de Kreweras

On considère la fonction g ci-dessous.

```
def g(n,m):
    assert type(n) is int
    assert type(m) is int
    assert 0 <= m and m <= n
    if n == 0 and m == 0:
        c = 1
    else:
        if m == 0: c = 0
        else:
            c = 0
            for i in range(1,m+1):
                c = c + g(n-1,n-i)
    return c
```

1. Calculer toutes les valeurs possibles de $g(n,m)$ pour $n \in [0,6]$.

```
>>> for n in range(7) :
        print 'n =', n, ':',
        for m in range(n+1) :
            print kreweras(n,m),
        print


n = 0 : 1
n = 1 : 0 1
n = 2 : 0 1 1
n = 3 : 0 1 2 2
n = 4 : 0 2 4 5 5
n = 5 : 0 5 10 14 16 16
n = 6 : 0 16 32 46 56 61 61
```

2. Vérifier que `12.*g(5,5)/g(6,6)` est une bonne approximation de π.

```
>>> 2.*6*g(5,5)/g(6,6)
3.1475409836065573
>>> 2.*12*g(11,11)/g(12,12)
3.1416005461074121
```

Portée des variables

Remarque C.7 : *Voir TD 3.15 page 130.*

On considère les fonctions `f`, `g` et `h` suivantes :

```
def f(x):
    x = 2*x
    print 'f', x
    return x
```

```
def g(x):
    x = 4*f(x)
    print 'g', x
    return x
```

```
def h(x):
    x = 3*g(f(x))
    print 'h', x
    return x
```

Qu'affichent les appels suivants ?

1.
```
>>> x = 2
>>> print x
2
>>> y = f(x)
f 4
>>> print x
2
>>> z = g(x)
f 4
g 16
>>> print x
2
>>> t = h(x)
f 4
f 8
g 32
h 96
>>> print x
2
```

2.
```
>>> x = 2
>>> print x
2
>>> x = f(x)
f 4
>>> print x
4
>>> x = g(x)
f 8
g 32
>>> print x
32
>>> x = h(x)
f 64
f 128
g 512
h 1536
>>> print x
1536
```

Exécution d'une fonction itérative

Remarque C.8 : *Voir TD 3.18 page 131.*

On considère la procédure `f` définie ci-dessous.

```
def f(n):
    for p in range(n+1):
        num = 1
        den = 1
        for i in range(1,p+1) :
            num = num*(n-i+1)
            den = den*i
        c = num/den
        print c,  #------------ affichage
    print         #------------ affichage
    return
```

1. Qu'affiche l'instruction `for n in range(7) : f(n)` ?

```
>>> for n in range(7) : f(n)

1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1
1 6 15 20 15 6 1
```

2. Que représente `c` à la fin de chaque itération sur `p` ?

Le $p^{ième}$ coefficient du binôme $(x + y)^n$ avec $0 \leq p \leq n$.

Index

Définitions

Exercices

Algorithmes

Liste des figures

Liste des exemples

Liste des exercices

Références

[1] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure et interprétation des programmes informatiques*. InterEditions, 1989.

Autre approche de l'algorithmique fondée sur la programmation fonctionnelle.

cité page 2.

[2] Jean-Luc Chabert, Evelyne Barbin, Michel Guillemot, Anne Michel-Pajus, Jacques Borowczyk, Ahmed Djebbar, and Jean-Claude Martzloff. *Histoire d'algorithmes : du caillou à la puce*. Belin, 1994.

Regard historique sur quelques algorithmes dont celui de la multiplication arabe.

cité page 23.

[3] Henri-Pierre Charles. *Initiation à l'informatique*. Eyrolles, 2000.

Petit livre d'introduction à l'algorithmique.

cité page 11.

[4] Karine Chemla and Guo Shuchun. *Les neuf chapitres : le classique mathématique de la Chine ancienne et ses commentaires*. Dunod, 2004.

Traduction française d'un texte chinois vieux de 2000 ans : « Jiuzhang suanshu », dont le 8ème chapitre, sous le titre « Fang cheng » (*la disposition rectangulaire*) décrit la méthode d'élimination de « Gauss ».

cité page 204.

[5] Christophe Darmangeat. *Algorithmique et programmation pour non-matheux*. Université Paris 7, www.pise.info/algo.

Cours en ligne d'initiation à l'algorithmique.

cité pages 18,61.

[6] Jean-Paul Delahaye. *Complexités : aux limites des mathématiques et de l'informatique*. Belin, 2006.

Ouvrage de vulgarisation sur la complexité.
cité page 21.

[7] Fernand Didier. Elaboration d'algorithmes itératifs
, iml.univ-mrs.fr/~lafont/licence/prog2.pdf.
Petit polycopié qui introduit simplement la notion d'invariant de boucle.
cité page 95.

[8] Pierre Florent, Michelle Lauton, and Gérard Lauton. *Outils et modèles mathématiques*, volume 3. Vuibert, 1977.
Ouvrage d'introduction à l'algèbre linéaire dont le chapitre 4 traite de la *résolution numérique des systèmes algébriques linéaires non homogènes.*
cité pages 169,200.

[9] Christine Froidevaux, Marie-Christine Gaudel, and Michèle Soria. *Types de données et algorithmes*. McGraw-Hill, 1990.
Bonne introduction aux types abstraits de données et à la complexité des algorithmes.
cité pages 172,173.

[10] Richard Gruet. Python 2.5 quick reference, rgruet.free.fr/#QuickRef, 2007.
Référence bien pratique qu'il faut avoir avec soi quand on programme en Python.
cité pages 41,57,92,153,155,194,198.

[11] Marc Guyomard. Spécification et programmation : le cas de la construction de boucles, www.irisa.fr/cordial/mguyomar/poly-init-06-07.ps, 2005.
Polycopié de cours qui aborde en détail la construction des boucles à partir de la notion d'invariant.
cité page 95.

[12] Donald Ervin Knuth. *The art of computer programming*, volume 1-Fundamental algorithms, 2-Seminumerical Algorithms, 3-Searching and sorting. Addison-Wesley, 1997.
« La » référence en algorithmique. Deux autres volumes sont en préparation : 4-Combinatorial Algorithms et 5-Syntactic Algorithms. Pour la petite histoire, D.E. Knuth est également l'auteur de TEX (prononcer tech comme dans technologie), système logiciel de composition de documents — utilisé pour générer le document que vous êtes en train de lire — créé à partir de 1977 pour répondre à la mauvaise qualité typographique de la deuxième édition de *The art of computer programming.*
cité pages 172,173.

[13] Zohar Manna and Richard Waldinger. *The logical basis for computer programming*, volume 1 : Deductive reasoning. Addison-Wesley, 1985.

Ouvrage sur les types abstraits de données présentés à l'aide d'un formalisme logique rigoureux.
cité pages 159, 193.

[14] Jean-Jacques Rousseau. *Physique et simulations numériques*. Université du Maine, site : `www.univ-lemans.fr/enseignements/physique/02/`, 2007.
Site pédagogique présentant plus de 300 simulations numériques de petits travaux pratiques de physique.
cité page 200.

[15] Jacques Rouxel. *Les Shadoks : ga bu zo meu*. Circonflexe, 2000.
Quand les Shadoks comptaient en base 4... BD issue d'une célèbre série télévisée d'animation française en 208 épisodes de 2 à 3 minutes, diffusée entre 1968 et 1973 (trois premières saisons) et à partir de janvier 2000 (quatrième saison).
cité page 25.

[16] Gérard Swinnen. *Apprendre à programmer avec Python*. O'Reilly, 2005.
Manuel d'initiation à la programmation basé sur PYTHON. Ce manuel existe dorénavant sous deux formes : un ouvrage imprimé publié par les éditions O'Reilly et un fichier PDF ou SXW librement téléchargeable sur `www.cifen.ulg.ac.be/inforef/swi/python.htm`
cité page 10.